Beuth Hochschule für Technik Berlin

Fachbereich Informatik und Medien

Studiengang Technische Informatik - Embedded Systems (M. Eng.)

Integration und Evaluierung eines Testboards zur Echtzeit-3D-Verarbeitung

Masterarbeit

zur Erlangung des Grades eines Master of Engineering

eingereicht von: Christian Konrad Elmar Schmiedl

Matrikel-Nr.: 797958

am: 02.03.2015

Erstgutachter: Dr. Anko Börner

Zweitgutachter: Prof. Dr. Volker Sommer

Kurzfassung

Eine der Herausforderungen im Bereich des Maschinellen Sehens ist es, räumliche Tiefe mithilfe von zweidimensionalen Kamerabildern zu ermitteln. Ein häufig verwendeter Ansatz hierbei basiert auf binokularer Disparität. In meiner Masterarbeit entwickle ich ein mobiles eingebettetes System zur Echtzeit-3D-Verarbeitung nach diesem Prinzip. Es nutzt eine Implementierung des Semi-Global Matching (SGM) Algorithmus auf einem FPGA.

Abstract

In the field of computer vision, one challenge is estimating spatial depth from 2D camera images. A common approach for this is using binocular disparity. With my master thesis I am providing a mobile embedded real-time computer stereo vision system. It utilizes an implementation of semi-global matching (SGM) on a FPGA.

Inhalt

1 Einführung und Zielsetzung

1.1 Motivation für den Einsatz eines Echtzeit-3D-Verarbeitungssystems

Für die Wahrnehmung unserer Umwelt nutzen Menschen unterschiedliche Sinne. Besonders der Sehsinn, auch visuelles System genannt, ermöglicht uns „eine genaue, sehr detaillierte, dreidimensionale Wahrnehmung“ (1 S. 166) unserer Umgebung. Diese Fähigkeit des Menschen auf ein Rechnersystem zu übertragen ist insbesondere für eingebettete Computersysteme von großer Relevanz. Man spricht hierbei von *maschinellem Sehen*. Autonome Kraftfahrzeuge können damit beispielsweise die vor ihnen liegende Straße und sich möglicherweise darauf befindende Hindernisse erkennen und entsprechend reagieren. Ein weiteres Beispiel sind spezielle Geräte die, unter Zuhilfenahme weiterer Sensoren, Innen- und Außenräume vermessen und kartieren können.

Das menschliche Auge ist in seiner Wahrnehmung auf Licht „zwischen 380 und 760 Nanometern“ (1 S. 168) beschränkt. Bei Computersystemen werden zur Bilderstellung Kameras verwendet. Der Messbereich der darin verbauten fotoaktiven Sensoren liegt nicht zwangsläufig im Bereich des von Menschen wahrnehmbaren Lichtes. Ein entsprechendes Computersystem kann folglich optische Signale erfassen, die dem menschlichen Betrachter nicht sichtbar sind.

Maschinelles Sehen kann dem menschlichen Sehen also überlegen sein. Jedoch beispielsweise bei der dreidimensionalen Wahrnehmung, oft auch Raumwahrnehmung genannt, hinken aktuelle Computersysteme dem visuellen System des Menschen hinterher. Die meisten Ansätze sind zu rechen- oder speicheraufwändig, um sie mit entsprechender Genauigkeit in Echtzeit durchzuführen. Einige verfahren jedoch erreichen Bildraten, die mit der des menschlichen Auges vergleichbar sind.

Ein Beispiel dafür ist der von Heiko Hirschmüller entwickelte, von einem externen Dienstleister auf einem FPGA implementierte und von mir in dieser Arbeit verwendete Algorithmus „Semi-Global Matching“ (2), den ich in Abschnitt 2.2 genauer erkläre. Er nutzt das Prinzip der binokularen Stereovision, welches ich in Abschnitt 2.1 beschreibe.

1.2 Zielsetzung im Kontext des assoziierten Forschungsprojektes

Diese Abschlussarbeit ist als Teil eines Forschungsprojektes am Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR)[1] entstanden. In dem Projekt wird ein System zur dreidimensionalen Erfassung der Umgebung und der Trajektorie, auf der es sich durch diese bewegt, entwickelt. Es vereint hierzu mehrere verschiedene Sensorsysteme und wird als „Integral Positioning System (IPS)“ (3 S. 21) bezeichnet. Die beiden wesentlichen Sensoren sind ein Inertialmesssytem (IMU) und zwei Videokameras, die als Stereokamera fungieren. Das System ist für den Innen- und Außenbereich[2] konzipiert.

Diese Kameras stellen ein passives System dar, da sie lediglich die durch die Umgebungslichter beleuchtete Umwelt erfassen. Eine häufig eingesetzte Alternative zur Erzeugung von Tiefenbildern mit einer Stereokamera stellen aktive Systeme, wie beispielsweise Radar und Laserscanner dar. Ebenso gibt es Einzelkamerasysteme, die ein bestimmtes Muster auf die zu messende Umgebung projizieren und aus der Verzeichnung desselbigen eine Tiefenschätzung vornehmen. Ein entscheidender Nachteil solcher aktiver Systeme ist, dass sie sich bei gleichzeitigem

---

[1] “Das DLR ist das Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt. Seine Forschungs- und Entwicklungsarbeiten in Luftfahrt, Raumfahrt, Energie, Verkehr und Sicherheit sind in nationale und internationale Kooperationen eingebunden. Darüber hinaus ist das DLR im Auftrag der Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig.“ (17)

[2] “Integral Positioning System(IPS) can be applied for indoor environments and outdoor environments” (3 S. 21)

Einsatz gegenseitig stören. Radarsysteme besitzen außerdem, trotz hoher Auflösung in der Entfernungsmessung, eine ansonsten geringe räumliche Auflösung. Dies führt dazu, „dass Informationen über Form und Größe des Objektes nicht abgeleitet werden können“ (4 S. 85).

Bei Laserscannern besteht das Problem, dass vor allem an Kanten „die Laserstrahlen stark gestreut“ (4 S. 86) werden, was zu verfälschten Messergebnissen führt. Ebenso führt besonders bei großen zu messenden Distanzen die möglicherweise nicht ausreichende Leuchtkraft des Laserstrahls dazu, „dass nicht genug Energie zum Sensor zurückkehrt“ (4 S. 86). Ebenso findet in diesem Fall eine „Aufweitung des Laserstrahls“ (4 S. 86) statt, „wodurch eine Punktmessung mehrere Objekte erfasst“ (4 S. 86). Kamerasysteme verfügen über eine „hohen räumlichen Auflösung im Megapixel-Bereich“ (4 S. 86) und „eine hohe zeitliche Auflösung“ (4 S. 86). Beispielsweise verfügten sie im Jahr 2013 über „eine Ausgangsdatenrate von bis zu 50 Msamples/s“[3] (4 S. 87), wohingegen „[d]er schnellste Laserscanner […] maximal 4 Msamples/s“ (4 S. 87) liefert.

Aktive Systeme haben jedoch auch Vorteile. So existieren Systeme, „die durch Laufzeitmessung oder mittels Interferometrie Entfernung aktiv messen und hierbei punktweise eine bessere Genauigkeit als Kameras erreichen“ (4 S. 84f). Auch Laserscanner und Radarsysteme haben Vorteile gegenüber passiven Kamerasystemen. Beispielsweise „können [sie] in vollständiger Dunkelheit weiterhin Objekte detektieren“ (4 S. 85). Speziell Radarsysteme können „[a]ufgrund des Dopplereffekts […] zudem sehr genau die relative Geschwindigkeit der beobachteten Objekte messen“ (4 S. 85). Eine Kombination mehrerer messtechnischer Methoden könnte möglicherweise die jeweiligen Nachteile der einzelnen verwendeten Systeme aufwiegen.

---

[3] Die Abkürzung *Msamples/s* bedeutet *millionen Messungen pro Sekunde*

Die Erstellung der Tiefenbilder erfolgt bei einem Stereokamerasystem aus dem Wissen um die Anordnung der Kameras und der relativen Verschiebung der einzelnen Bildpunkte beider Aufnahmen. Sie können mit dem bestehenden System nicht im laufenden Betrieb, sondern erst im Nachhinein, berechnet werden. Das bedeutet, dass das mobile IPS zuerst in einem Messdurchgang Daten aufnimmt und anschließend an einen stationären Computer angeschlossen wird, dem es diese übermittelt. Dieser Computer errechnet das Tiefenbild, fusioniert daraufhin alle ermittelten Informationen und berechnet daraus die Trajektorie.

Meine Abschlussarbeit hebt diese zeitliche Einschränkung an das IPS auf. Sie ermöglicht es, die Tiefenkarte im laufenden Betrieb zu erstellen und sich folglich in ihr zu lokalisieren. Dadurch stehen die Messergebnisse schon während eines Versuches direkt vor Ort zur Verfügung.

Besonders die robuste Erkennung und gegenseitige Zuordnung einzelner Merkmale im linken und rechten Bild stellt eine große Herausforderung dar. Mit zunehmender Kameraauflösung steigen der dafür erforderliche Rechenaufwand und der entsprechende Speicherplatzbedarf stark an. Außerdem müssen, neben der algorithmischen Verwertung der Bildinformationen, große Datenmengen schnell zwischen den einzelnen verarbeitenden Systemen übertragen werden.

2 Relevante Grundlagen der Bildverarbeitung

2.1 Grundlagen der Stereovision in der Bildverarbeitung

Die Bildverarbeitung beschäftigt sich, als Teilgebiet der Informatik, mit der Verarbeitung von Signalen, welche mit fotografischen Mitteln gemessen wurden. Ihr Ziel ist die Extraktion von Informationen aus Bilddaten. Beispiele für Bildverarbeitungsprozesse sind Kantenerkennung, Rauschreduktion, Merkmalsextraktion und Stereomatching.

In dieser Arbeit erzeuge ich aus dem Bild zweier zueinander räumlich versetzter, gleich ausgerichteter Kameras eine dreidimensionale Schätzung der Umgebung. Dies entspricht der binokularen Raumwahrnehmung des Menschen und nutzt das Prinzip der „Querdisparität (auch: binokulare Disparität)“ (5 S. 192). Die Disparität gibt die Differenz der Koordinaten zweier demselben Objektpunkt entsprechender Bildpunkte im linken und rechten Kamerabild an. Je näher sich das Objekt an den Kameras befindet, desto größer ist dieser Unterschied. Die beiden Bildpunkte haben in diesem Fall eine große Disparität. Je weiter der Objektpunkt entfernt ist, desto kleiner ist die binokulare Disparität, also die horizontale Verschiebung, zwischen seinen beiden Bildpunkten.

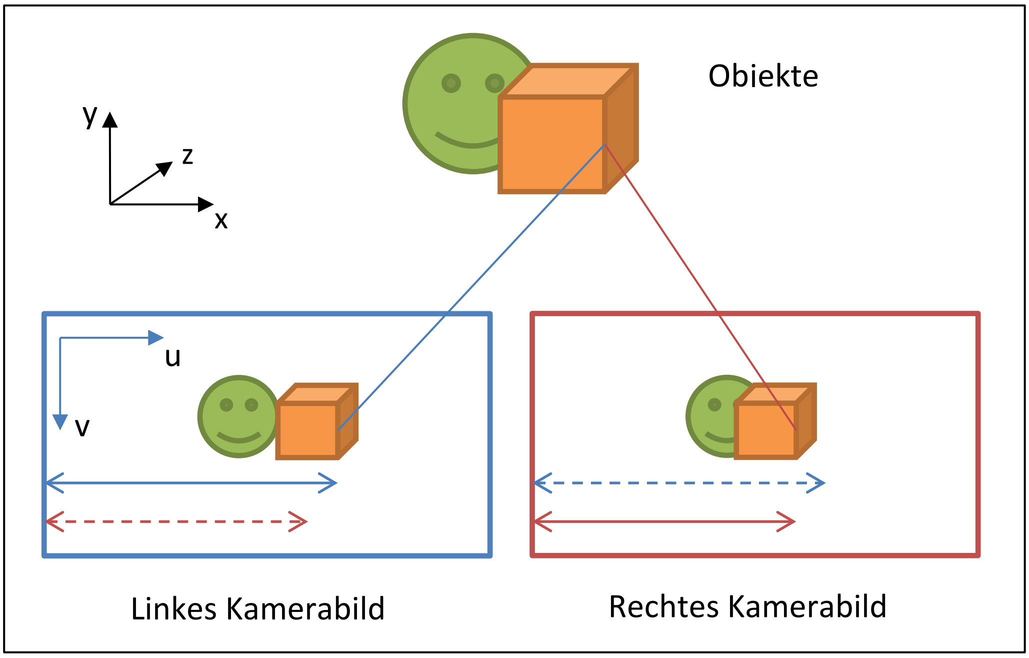


Abbildung 1 – Binokulare Disparität

In Abbildung 1 ist das Prinzip der binokularen Disparität dargestellt. Die Koordinaten im dreidimensionalen Objektraum sind mit *x* für die Horizontale, *y* für die Senkrechte und *z* für die Tiefe angegeben. Um Verwechselungen zu vermeiden, verwende ich im zweidimensionalen Bildraum statt der Koordinaten *x* und *y* die Bezeichnungen *u* für die horizon-

tale und *v* für die vertikale Achse. Dabei ist zu beachten, dass die *v*-Koordinate im Gegensatz zur *y*-Achse von oben nach unten wächst[4].

Im Objektraum liegt das Gesicht weiter hinten, als der Würfel. Betrachten wir beispielsweise den Mittelpunkt der vom Betrachter aus vorderen rechten Kante des Würfels. Dieser ist durch Linien vom Objektraum in den linken und rechten Bildraum gekennzeichnet. Durch jeweils zwei Pfeile in den Kameraebenen habe ich die *u*-Koordinate des Bildpunktes markiert. Der gestrichelte Pfeil ist dabei die *u*-Koordinate des entsprechenden Punktes im jeweils anderen Bild. Die Differenz der beiden Pfeile entspricht der Disparität unseres Beispielpunktes. Die Herausforderung hierbei liegt darin, aus den beiden Kamerabildern zu erkennen, welche zwei Punkte im Bildraum einem gemeinsamen Punkt im Objektraum zugeordnet sind.

Beim Stereomatching wird jedem[5] Bildpunkt des einen Kamerabildes ein korrespondierender Punkt im jeweils anderen Bild zugeordnet. Das Ergebnis wird als Disparitätsbild bezeichnet und kann als Abbildungsfunktion von dem einen in das andere Kamerabild verstanden werden. Hierbei treten verschiedene Effekte auf, die besondere Herausforderungen für die Berechnung der Disparität darstellen. Ich unterscheide im Folgenden zwischen Effekten im Objektraum, in der Kamera und in dem verwendeten Algorithmus.

Im Objektraum können beispielsweise Verdeckungen einzelner Bildbereiche auftreten, sodass diese sind nur in einem der beiden Bilder zu sehen sind. Ein Beispiel dafür ist das vom Betrachter aus gesehen rechte Auge des Gesichtes in Abbildung 1, das im rechten Kamerabild von dem Würfel verdeckt wird. Weitere Herausforderungen im Objektraum stellen Reflektionen, Beleuchtungsunterschiede, fehlende Texturen und

[4] Vgl. “Der Koordinatenursprung des Bildkoordinatensystems liegt per Definition in der oberen linken Bildecke.” (4 S. 14)
[5] Man spricht in diesem Fall auch von „dense stereo matching“ (2 S. 1).

Mehrdeutigkeiten dar. Bei Betrachtung einer untexturierten, weißen Wand existieren keine Merkmale, die eine Zuordnung korrespondierender Bildpunkte ermöglichen. Auch wenn sich gleichartige Merkmale auf einer größeren Fläche häufig wiederholen, wie beispielsweise bei einer geziegelten Hauswand, ist die eindeutige Zuordnung von Pixeln im linken und rechten Bild problematisch.

In der Kamera führt die Vignettierung, also die durch das Kameraobjektiv hervorgerufenen Verdunkelung des Bildes in den Randbereichen, dazu, dass korrespondierende Punkte nicht unbedingt die gleiche Helligkeit im Bildraum ausweisen. Auch die Verzerrung des Bildes durch die abbildende Optik erschwert das Stereomatching. Abhängig vom verwendeten Algorithmus können verschiedene weitere Effekte auftreten. Bei der „scanline optimization“ (6 S. 175) kommt es beispielsweise zur Bildung von Artefakten in Form von Streifen im Bild[6].

Das Disparitätsbild gibt lediglich Auskunft darüber, um wie viele Pixel zwei korrespondierende Punkte zueinander verschoben sind. Um ein echtes Tiefenbild im dreidimensionalen euklidischen Objektraum zu berechnen, benötigt man weitere Informationen über das Kamerasystem. Die Orientierung der Kameras zueinander und die Verzeichnung durch die verwendeten Objektive sind hierbei die wichtigsten Parameter. Um diese zu ermitteln, wird das Kamerasystem kalibriert.

[6] Vgl. “streaking artefacts are inherent in scanline optimization” (6 S. 175)

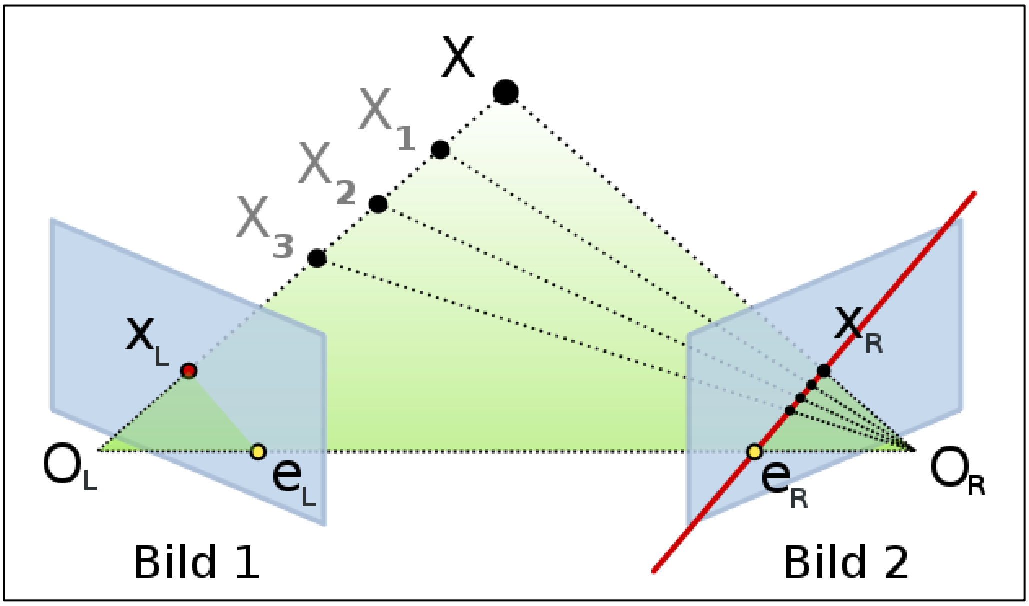


Abbildung 2 – „Schematische Darstellung der Epipolargeometrie“ (21) [7]

Ein kalibriertes Kamerasystem hat auch bei der Berechnung des Disparitätsbildes den Vorteil, dass korrespondierende Punkte nur auf sogenannten Epipolarlinien gesucht werden müssen. Die Epipolarlinie eines Punktes in einem Bild bezeichnet die Menge der Punkte innerhalb des anderen Bildes, innerhalb der der ihm entsprechende Bildpunkt liegt. In Abbildung 2 ist eine solche Epipolarlinie dargestellt. Der Objektpunkt X wird in *Bild 1* auf den Bildpunkt X_L abgebildet. Die Objektpunkte X_1, X_2 und X_3 würden auf denselben Punkt abgebildet werden. Im rechten Bild ergeben die Objektpunkte X_1, X_2 und X_3 jedoch Bildpunkte, die vom dort tatsächlich aufgenommenen Bildpunkt X_R abweichen, jedoch alle auf der rot eingezeichneten Epipolarlinie liegen.

Wenn die beiden Bildebenen gleich ausgerichtet sind, in einer Ebene liegen und etwaige Verzeichnungen korrigiert wurden, spricht man von rektifizierten Bildern. Diese können unter Kenntnis der Kamerageometrie berechnet werden. Unter der Annahme, dass sich die Kamerageometrie im laufenden Betrieb nicht verändert, kann die Rektifizierungsabbildung einmalig berechnet und als Look-Up-Tabelle gespeichert

[7] Abbildung 2 habe ich der Online-Enzyklopädie Wikipedia entnommen. (21)

werden. Die Rektifizierung der Bilder vor dem Stereomatching vereinfacht die Suche nach korrespondierenden Punkten. „Die Epipolarlinie verläuft dann entlang der [u]-Achse, wodurch erhebliche Vorteile bei der Verarbeitung durch den linearen Pixelzugriff zu erwarten sind.“ (4 S. 17)

2.2 Der Ablauf des Semi-Global Matching (SGM) Algorithmus

Ein Verfahren zum Erzeugen des Disparitätsbildes aus zwei Kamerabildern stellt der Semi-Global Matching (SGM) Algorithmus dar. Bei der Zuordnung korrespondierender Pixel werden Informationen sowohl aus ihrer direkten (lokalen) Umgebung, als auch aus dem gesamten (globalen) Bild verwendet.

Dabei wird angenommen, dass dem Bild eine gewisse Glattheit innewohnt. Das bedeutet, dass sich die Disparität nicht zwischen allen Pixeln beliebig sprunghaft ändert, sondern dass relativ große zusammenhängende Flächen annähernd gleicher Disparität vorliegen. Die Berechnung des Disparitätsbildes wird, wie in Gleichung (1), als Energiefunktion[8] *E(D)* formuliert. Ihr globales Minimum erreicht diese Funktion für das gesuchte Disparitätsbild *D*.

$$E(D) = \sum_{p} \left(C(p, D_p) + \sum_{q \in N_p} P_1 T[|D_p - D_q| = 1] + \sum_{q \in N_p} P_2 T[|D_p - D_q| > 1] \right) \quad (1)$$

Der erste Term, die Funktion *C(p,d)*, gibt dabei die Kosten an, die anfallen, um den Pixel *p* aus dem einen Bild dem entsprechenden, um Disparität *d* verschobenen, Pixel *q* aus dem anderen Bild zuzuordnen. Der zweite und dritte Term erzeugen dabei zusätzliche Kosten P_1 und P_2 für

[8] Vgl. “The pixelwise cost and the smoothness constraints are expressed by defining the energy E(D) that depends on the disparity image D.” (2 S. 3)

Disparitätsänderungen[9]. Kleine Änderungen werden dabei weniger stark bestraft als große. Daher sollte P_2 stets größer oder gleich P_1 sein[10]. Häufig sind Disparitätsunterschiede auch als Unterschiede in der Intensität der entsprechenden Pixel sichtbar. Dies wird dadurch ausgenutzt, dass P_2 an den Intensitätsunterschied zwischen den Pixel *p* und *q* angepasst wird[11].

Zwischen dem linken und rechten Kamerabild können radiometrische Differenzen auftreten. Diese entstehen beispielsweise durch Vignettierung, verschiedene Belichtungszeiten und blickwinkelabhängige Reflektionen[12]. Zur Berechnung der Matchingkosten *C* gibt es verschiedene Methoden. Verfahren wie Mutual Information (MI) und Census gelten als relativ robust gegenüber solcher radiometrischer Differenzen[13].

Die SGM-Implementierung auf einer FPGA-Karte, die ich in dieser Arbeit nutze, verwendet zur Kostenberechnung die Census-Methode. Dabei wird für jeden Pixel in einer Reihe von Bits gespeichert, ob die Pixel in seiner Nachbarschaft einen kleineren Wert, als dieser zentrale Pixel, haben oder nicht[14]. Die Matchingkosten zweier Pixel entsprechen dabei

---

[9] Vgl. "The second term adds a constant penalty P_1 for all pixels q in the neighborhood N_p of p, for which the disparity changes a little bit (i.e. 1 pixel). The third term adds a larger constant penalty P_2, for all larger disparity changes." (2 S. 3)
[10] Vgl. "a constant penalty P_1 […] larger constant penalty P_2," (2 S. 3)
[11] Vgl. "Discontinuities are often visible as intensity changes. This is exploited by adapting P_2 to the intensity gradient, i.e. $P_2 = \frac{P_2'}{|I_{b\mathbf{p}} - I_{b\mathbf{q}}|}$" (2 S. 3)

[12] Vgl. "Radiometric differences often occur due to […] the vignetting effect, different exposure times, […] reflection, which is viewpoint dependent […] etc" (12 S. 174)
[13] Vgl. "An extensive study of different matching costs […] showed that MI can model all global radiometric differences as well as image noise very well, but it degrades with increasing local radiometric differences […]. The same study identified Census as the most robust matching cost for stereo vision." (12 S. 175)
[14] Vgl. "Census […] encodes the local neighborhood […] around each pixel into a bit vector that only sotres if the compared, neighboring pixel has a lower value than the center pixel or not." (12 S. 175)

dem Hammingabstand, also der Anzahl der nicht übereinstimmenden Bits, der beiden so ermittelten Bitreihen[15].

Die Suche nach dem Disparitätsbild, für welches die Energiefunktion in Gleichung (1) minimal ist, wird Optimierung dieser Funktion genannt. Diese wird nicht zweidimensional, sondern eindimensional, also entlang einzelner Pfade vorgenommen, da sie sonst ein NP-vollständiges Problem darstellen würde[16]. Dabei werden beim SGM mehrere Pfade symmetrisch aus allen Richtungen des Bildes verfolgt[17]. Für jeden Pixel und jede Disparität werden die Kosten entlang dieser Pfade berechnet und anschließend die Disparität mit den geringsten Kosten gewählt.

3 Die angewandte Methodik

3.1 Die Methodik beim Systementwurf

Zur erfolgreichen Entwicklung eines Systems ist es hilfreich, von Anfang an Anforderungen festzulegen, die sich aus den zuvor definierten Projektzielen ableiten. Die Formulierung der Anforderungen hilft dabei, schon früh eine konkrete Vorstellung des zu entwickelnden Systems zu erhalten. Ebenso dienen sie während der gesamten Projektlaufzeit als Orientierungshilfe. Beispielsweise helfen sie bei der Auswahl einer geeigneten Technologie aus mehreren zur Verfügung stehenden Alternativen. Eine wesentliche Eigenschaft klar formulierter Anforderungen ist auch, dass sich aus ihnen Testszenarien, sowohl für das Gesamtsystem, als auch für einzelne Komponenten, ableiten lassen. Damit kann die jeweilige Komponente schon vor Fertigstellung des endgültigen Systems

[15] Vgl. "Pixel-wire matching is done by computing the Hamming distance of bit vectors of corresponding pixels." (12 S. 175)
[16] Vgl. "optimization is performed in 1D only […] not in 2D, which is NP complete." (12 S. 175)
[17] Vgl. "The novel idea of SGM is the computation along several paths, symmetrically from all directions through the image." (12 S. 175)

getestet und im Erfolgsfall als fehlerfrei betrachtet und eingesetzt werden.

Zur Umsetzung der Anforderungen gibt es in der Regel mehrere alternative Lösungen. Ich habe in der Regel zwei oder drei davon eingehend betrachtet und anschließend eine ausgewählt. Ein wichtiges Kriterium hierbei war, neben der Leistungsfähigkeit einer möglichen Lösung, ihre Komplexität und der damit verbundene Implementierungsaufwand. In vielen Projekten, in denen ich bisher gearbeitet habe, wurden auch noch nach der Erfassung der Anforderungen einzelne solche geändert oder hinzugefügt. Dies geschieht zum Beispiel durch neue Erkenntnisse oder durch Fehler bei der Formulierung bestehender Anforderungen. Daher habe ich bei der Wahl der Umsetzung auch großen Wert auf ihre spätere Anpassbarkeit gelegt. Dies bedeutet zum einen, dass Änderungen nicht grundsätzlich ausgeschlossen oder unnötig erschwert werden, und zum anderen, dass auch Dritte sich leicht in das System einarbeiten und etwaige Änderungen vornehmen können.

Bei der Auswahl der verwendeten Hardware wurde ich von Dr. Maximilian Buder unterstützt. Gemeinsam haben wir die groben Randbedingungen und Anforderungen an das Hardwaresystem festgelegt, woraufhin er hat sich um die Auswahl und Beschaffung der Einzelkomponenten gekümmert hat. Vor Inbetriebnahme der Hardware im eingebetteten System habe ich diese auf einem Arbeitsplatzrechner getestet.

3.2 Das Vorgehen bei Softwareentwicklung und -test

Ein strukturiertes Vorgehen bei der Programmerstellung ist Grundlage für eine erfolgreiche Entwicklung des geforderten Systems. Bevor ich eine Funktionalität implementiert habe, habe ich jeweils verschiedene Lösungsansätze untersucht. Durch Internetrecherche und im Gespräch mit Kollegen habe ich hierbei neue Ideen gesammelt und mit meinen

eigenen Vorstellungen abgeglichen. Gegebenenfalls habe ich mehrere Lösungswege in separaten, prototypischen Programmen implementiert, um sie besser verstehen und ihre Vor- und Nachteile besser abschätzen zu können.

An einigen Stellen habe ich mich im an bestehendem Quelltext aus den internen Softwarebibliotheken des DLR, wie beispielsweise der *OSLib* und der *SFSLib* orientiert. Beispielsweise ist das Grundgerüst, welches ich für den *SGMOPFeeder* genutzt habe, dem *StereoFeeder* der *SFSLib* entnommen. Ein anderes Beispiel sind die in der *OSLib* enthaltenen Testfälle, welche ein gutes Bild über die Verwendung der jeweiligen, in ihnen verifizierten, Klassen vermitteln. In einigen Fällen habe ich auch im Internet nach verschiedenen Ideen zur Lösung algorithmischer Probleme gesucht. So basiert die farbliche Visualisierung der Dispariätsbilder auf einer Seite der Online-Enzyklopädie Wikipedia. Den Link zu der entsprechenden Webseite habe ich im Quelltext der Klasse *SGMOPFeeder* in der Methode *disparityToColor* als Kommentar hinterlegt.

Jede dem Programm hinzugefügte Funktionalität stellt einen weiteren Entwicklungsschritt dar, welchen ich, nach Fertigstellung, nicht nur als Einzelkomponente, sondern als auch im Kontext des Gesamtsystems getestet habe. Dies führte dazu, dass das zu entwickelnde System stets operabel war. Die Lauffähigkeit war natürlich nur unter der Einschränkung reduzierter Funktionalität gegeben, da zum jeweiligen Zeitpunkt noch nicht alle Anforderungen umgesetzt waren.

Durch Einhalten einiger Programmierrichtlinien kann man verhindern, dass der implementierte Quelltext unübersichtlich und somit unwartbar wird. Aussagekräftige Namen für Bezeichner von Variablen und Funktionen, weitestgehend kurze Funktionsrümpfe ohne große Verschachtelungstiefe und eine Aufteilung der Software in einzelne Komponenten sind Beispiele dafür. Die Kapselung der Software in Komponenten hat,

neben Gründen der Übersichtlichkeit, den Vorteil, dass diese unabhängig voneinander getestet werden können. Beispielsweise habe ich den threadsicheren Ringpuffer, den ich in Abschnitt 5.4.2 genauer beschreibe, unabhängig vom restlichen Programm entwickelt. In einem separaten Testfall habe ich verifiziert, dass sein Verhalten dem geforderten solchen entspricht. Häufig ist es ausreichend, den korrekten Ablauf eines Quelltextabschnittes mit dem Auslesen des Wertes einer Variablen zu überprüfen. Dies kann entweder durch textuelle Ausgabe auf die Konsole oder in eine Systemdatei, durch einen Debugger oder durch entsprechende Überprüfung im Quelltext geschehen. Letzteres bleibt in der Regel im späteren Quelltext als Fehlerbehandlung erhalten.

Damit der Leser der Quelltextes leicht erkennt, welchen Zweck eine Funktion beziehungsweise Variable erfüllt, habe ich ihr einen Kommentarblock vorangestellt, der ihr Verhalten beziehungsweise ihre Verwendung beschreibt. Darüber hinaus habe ich bei Funktionen, neben der Beschreibung der Ein- und Ausgabeparameter, gegebenenfalls erwähnt, welche Vorbedingungen die übergebenen Argumente erfüllen müssen und welche Einschränkungen für die zurückgegebenen Werte gelten.

3.3 Die Entwicklungsumgebung

Zur Entwicklung der Software habe ich verschiedene Werkzeuge genutzt. Eine wesentliche Rolle haben integrierte Entwicklungsumgebungen (IDEs) gespielt. Unter Windows habe ich Microsoft Visual Studio verwendet und unter Linux die quelloffene Software Eclipse mit entsprechender Erweiterung für C und C++.

In beiden IDEs ist ein visueller Debugger integriert. Dieser ermöglicht es, zur Laufzeit das Verhalten der entwickelten Software zu beobachten und zu überprüfen. Beispielsweise kann man das Programm automatisch anhalten lassen, wenn es eine bestimmte Quelltextzeile ausführt

oder eine Variable einen bestimmten Wert annimmt. Sobald das Programm angehalten wurde, kann man den aktuellen Programmzustand untersuchen und verändern. Speziell beim Debugging der Netzwerkübertragung habe ich außerdem Wireshark, ein Werkzeug zur Analyse von Netzwerkverkehr, genutzt.

Neben dem Debugging habe ich auch das Profiling der IDEs genutzt. Dabei wird zur Laufzeit gemessen, wie viele Ressourcen von dem Programm benötigt werden. Besonders interessant ist dabei die Information, welche Funktionen die meiste Rechenzeit benötigen. Mit den so gewonnenen Erkenntnissen konnte ich mich gezielt der Optimierung eben dieser Funktionen widmen und die damit verbundenen Flaschenhälse entfernen.

Ein weiteres Werkzeug bei der Programmierung ist eine Versionskontrollsoftware. Sie ermöglicht es, verschiedene Entwicklungspfade zu verfolgen und jeden in ihr erfassten Entwicklungszustand später wiederherzustellen. Diese Erfassung eines aktuellen Zustandes wird häufig auch als *Commit* bezeichnet. Dabei ist es meiner persönlichen Erfahrung nach besonders wichtig, jedem *Commit* eine aussagekräftige Nachricht beizufügen. Diese beschreibt kurz, aber vollständig, welche Änderungen seit dem letzten *Commit* vorgenommen wurden. Außerdem ist es hilfreich anzugeben, ob und mit welchem Ergebnis das Programm gerade einem möglichen Test unterzogen wurde. Ich habe die in meiner Arbeitsgruppe am DLR gängige Versionskontrollsoftware *SVN* genutzt. Unter Windows habe ich sie mit der grafischen Benutzeroberfläche *Tortoise* und unter Linux von der Kommandozeile aus bedient.

3.4 Die verwendete Fremdsoftware

Ich habe in meiner Arbeit verschiedene externe Softwarewerkzeuge und Bibliotheken genutzt. Damit die von mir entwickelte Software auch auf

andere Systeme übertragen und dort genutzt beziehungsweise verifiziert werden kann, ist es essentiell, dass bekannt ist, welche fremde Software ich in welchen Versionen genutzt habe.

Ich habe nach Möglichkeit keine betriebssystemspezifischen Bibliotheken, sondern eine in der Abteilung, in der ich meine Abschlussarbeit schreibe, entwickelte umfangreiche C++-Softwarebibliothek namens *OSLib* verwendet. Sie dient der plattformunabhängigen Implementierung von Programmen und wird intern als Standardbibliothek genutzt. Sie wird erweitert durch die *OSVisionLib*, welche darüber hinaus einige Funktionen zur Bildverarbeitung enthält. Die Übertragung und Synchronisation der Sensordaten innerhalb des Sensorkopfes erfolgt über die ebenfalls in dieser Abteilung entwickelte *SFSLib*[18]. Die jeweiligen Daten werden mit einem Zeitstempel versehen[19] und durch eine Reihe von gekapselten[20] Verarbeitungsschritten geleitet. Diese gekapselten Verarbeitungsschritte werden als *Feeder* bezeichnet[21] und verfügen über Eingänge und Ausgange[22] für die von ihnen zu verarbeitenden Daten. An diesen werden sie zu einem Netzwerk zusammengeschlossen, welches den gesamten Verarbeitungsprozess der Sensordaten beschreibt. Ein einfaches Beispiel dafür ist der *StereoFeeder*, der an zwei Eingängen je ein Bild annimmt und an einem Ausgang ein Bild bereitstellt, in dem die Eingangsbilder nebeneinander gelegt wurden. Ein weiteres Beispiel stellt der von mir entwickelte *SGMOPFeeder* dar, den ich in Abschnitt 5.1.1 genauer beschreibe.

[18] Die *SFSLib* ist eine „C++ Library for Sensor [Feeding] an[d] Synchronization“. Vgl. https://svn.dlr.de/SFSLib/trunk/SFSLib/README.TXT Revision 1934
[19] Vgl. “A high precision clock
generates timestamps to which all sensor communication is referenced.” (3 S. 23)
[20] Vgl. “a particular task is encapsulated in a container” (3 S. 23)
[21] Vgl. “those containers [are] called feeder[s]” (3 S. 23)
[22] Vgl. “having […] defined inputs and outputs” (3 S. 23)

Als Testumgebung und Benutzeroberfläche habe ich die Anwendung *IPSApp* verwendet, die Teil der *SFSLib* ist. Sie ermöglicht das Laden und Ausführen von in XML-Dateien definierten Feedernetzwerken.

Im Folgenden habe ich, unter Angabe der verwendeten Version, die wesentlichen von mir genutzten Werkzeuge und Bibliotheken aufgelistet.

- Betriebssysteme:
    - Windows 7 64bit
    - LUbuntu 14.04.1 LTS 64 Bit mit Kernel Version 3.13.0-39
- Entwicklungsumgebung:
    - Eclipse Luna Version 4.4.1
    - Visual Studio 2012
- C und C++ Compiler:
    - Visual Studio 2012 C++ Compiler (Windows)
    - GCC Version 4.8.2 (Linux)
- Bibliotheken:
    - OSLib SVN-Revision 5030
    - SFSLib SVN-Revision 1956
    - Embedded GNU C Library Version 2.19
    - GNU Standard C++ Library Version 4.8.2
    - FreeImage Library Version 3.15.4-3
- Versionskontrolle: SVN Version 1.8.8
- Buildtool[23]:
    - CMake Version 3.1.0 (Windows)
    - CMake Version 2.8.12.2 (Linux)
- Netzwerkanalyse: Wireshark Version 1.10.6
- CAD Software[24]: Autodesk Inventor Professional 2012

[23] Das Buildtool dient dem automatisierten Generieren von Projektdateien und Compileranweisungen.

- Bildverarbeitung:
    - IrfanView Version 4.38
    - GIMP Version 2.8.14
- Bildauswertung:
    - OpenCV Version 2.4.10
    - Python Version 2.7.4
    - NumPy Version 1.9.1

4 Der Entwurf des eingebetteten Systems

4.1 Die Anforderungen an das eingebettete System

Das eingebettete System soll sowohl zur 3D-Vermessung im Gebäudeinneren, als auch im Außenbereich eingesetzt werden. Daher muss es im laufenden Betrieb portabel und mobil einsetzbar sein. Die Energieversorgung soll dabei über gängige Stromquellen möglich sein. Die Anbindung der Datenschnittstelle erfolgt mittels Gigabit-Ethernet, um möglichst hohe Datenraten zu erreichen.

Das System wird in das am DLR verwendete Framework zur Sensordatenverarbeitung integriert. Die Bilddaten und die Kamerageometrie werden darüber an das System übertragen. Das System soll vor dem Stereomatching eine Rektifizierung der Eingangsbilder durchführen. Dazu wird neben den Bilddaten auch die Kamerageometrie ausgewertet und übertragen. Die Rektifizierung erfolgt mittels einer Look-Up-Tabelle. Um zu verhindern, dass einzelne Pixel im rektifizierten Bild nicht mit Werten belegt werden, wird von dort aus jedem Pixel ein Wert zugeteilt, der aus dem nicht-rektifizierten Bild (Urbild) ermittelt wird. Hierbei wird die Umgebung des entsprechenden Pixels im Urbild bilinear interpoliert.

---

[24] Die CAD (Computer Aided Design) Software habe ich zum Entwurf eines Abstandhalters zwischen FPGA-Karte und Carrier-Board eingesetzt.

Das System soll weiche Echtzeitanforderungen erfüllen. Das bedeutet, dass die Verarbeitungsrate der Bilder im Mittel mindestens so hoch ist, wie die Aufnahmerate des Kamerasystems. Würden wir harte Echtzeit fordern, so müssten wir einen festen Zeitwert angeben, nach dem die Verarbeitung abgeschlossen ist. Sollte das System diesen Spitzenwert einmal übersteigen, würde die harte Echtzeitanforderung als nicht erfüllt gelten. Für die geforderte durchschnittliche Bildrate wird kein fester Wert vorgegeben, sie soll jedoch im ganzzahligen Hertzbereich liegen.

4.2 Der Aufbau des eingebetteten Systems

Das Gesamtsystem ist wie in Abbildung 3 dargestellt aufgebaut und besteht aus zwei Grundkomponenten. Die eine Komponente beinhaltet, neben weiteren Sensoren, die Kameras und das Framework zur Sensordatenverarbeitung. Diese Komponente bezeichne ich im Folgenden als Sensorkopf.

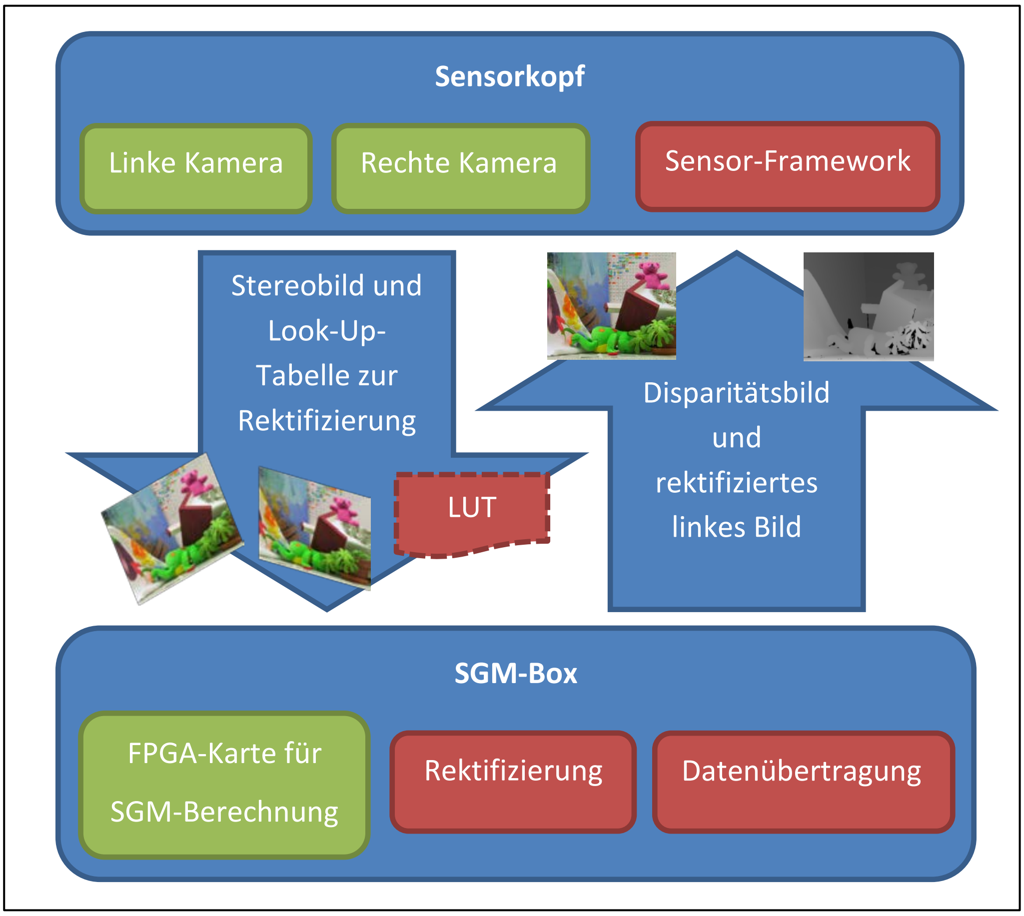


Abbildung 3 - Aufbau des eingebetteten Systems [25]

Die andere Komponente ist ein Rechner, der die FPGA-Karte zur Berechnung des SGM enthält. Dieser übernimmt neben der Berechnung der Disparitäten auch die Rektifizierung der Stereobilder. Diese Komponente bezeichne ich im Folgenden als SGM-Box.

Die beiden Komponenten sind per Gigabit-Ethernet miteinander verbunden. Der Sensorkopf sendet die Stereobilder an die SGM-Box. Beim ersten Bilderpaar berechnet er, basierend auf der Kamerageometrie, eine Look-Up-Tabelle zur Rektifizierung der Kamerabilder. Diese wird, zusammen mit dem ersten Bilderpaar jeder Verbindung, an die SGM-Box übertragen. Wir nehmen folglich an, dass sich weder die Kamerageometrie, noch die Auflösung der Bilder während der Verbindung ändert,

[25] Die in der Abbildung enthaltenen Stereo- und Disparitätsbilder sind dem Middlebury-Datensatz „Teddy“ (12) entnommen.

da die Look-Up-Tabelle sonst neu berechnet und übertragen werden müsste.

Die SGM-Box rektifiziert die empfangenen Stereobilder, formatiert sie neu und sendet sie an die FPGA-Karte. Das rektifizierte linke Kamerabild und das von der FPGA-Karte berechnete Disparitätsbild werden anschließend an den Sensorkopf übermittelt. Dort können sie gespeichert, dargestellt und weiterverarbeitet werden.

Das Sensor-Framework muss nicht notwendigerweise auf einem Rechner innerhalb des Sensorkopfes laufen, sondern kann auch auf einem externen Rechner ausgeführt werden. Dieser wird per Gigabit-Ethernet und USB 2.0 High-Speed angeschlossen. In diesem Dokument betrachte ich das Sensor-Framework der Einfachheit halber als Bestandteil des Sensorkopfes, obwohl ich es stets auf einem externen Rechner laufen lassen habe.

4.3 Die verwendeten Technologien

4.3.1 Die SGM-Box als eingebettetes System

4.3.1.1 Technische Daten der SGM-Box

Die SGM-Box stellt ein eigenständiges Rechnersystem dar, welches sich aus drei Platinen zusammensetzt. Die Hauptplatine enthält den Hauptprozessor (CPU) und den Arbeitsspeicher (RAM). Sie wird durch eine Peripherieplatine um einige standardisierte Anschlüsse erweitert. Über diese ist auch die FPGA-Karte angebunden, welche die dritte Platine darstellt. Ich habe sie in Abschnitt 4.3.2 separat beschrieben.

Bei der Hauptplatine handelt es sich um das Board *conga-TC87* von der congatec AG. Es ist im Formfaktor „COM Express® Compact, (95 x 95 mm), Type 6 Connector Layout“ (7 S. 1) gefertigt und beinhaltet eine CPU vom Typ „Intel® Core™ i7-4650U“ (7 S. 1) und zwei Sockel für Ar-

beitsspeicher vom Typ „SO-DIMM DDR3L“ (7 S. 1). In einen der beiden Sockel habe ich das RAM-Modul „DDR3L-SODIMM-1600 (8GB)“ (7 S. 2) eingesetzt. Den anderen Sockel habe ich für die Möglichkeit der späteren Erweiterung des Systems frei gelassen.

Als Peripherieplatine verwende ich das „COM Express® Type 6 Ultra Lite Carrier“ (8 S. 1) von der Connect Tech Inc., welches ich im Folgenden auch als *Carrier-Board* bezeichne. Neben der bereits erwähnten Erweiterung der Hauptplatine um einige Peripherieanschlüsse[26], enthält es Anschlüsse zur Spannungsversorgung. Diese stellt mit +12 Volt Gleichspannung[27] die Hauptenergieversorgung der SGM-Box dar, von der alle weiteren Spannungen abgezweigt und die anderen beiden Platinen betrieben werden. Im Systemstart habe ich einen kurzzeitigen Stromfluss von mehr als 3 Ampere gemessen. Ein Netzteil, welches bei 12 Volt einen Stromfluss von 4 Ampere zulässt und folglich eine Ausgangsleistung von 48 Watt besitzt, hat sich als stabil erwiesen.

Das Board enthält zwei kombinierte mSATA/miniPCIe-Slots. Das bedeutet, dass die Slots für beide Protokolle verwendet werden können. Die jeweilige Funktion wird durch Jumper gewählt[28]. An einen der beiden Slots ist via mSATA eine Solid-State Disk (SSD) vom Typ Crucial M500 mit 240 Gigabyte Speicherkapazität angeschlossen. Auf dieser ist das Betriebssystem und die von mir entwickelte Software der SGM-Box installiert. Der andere Slot dient der Anbindung der FPGA-Karte mittels miniPCIe.

Desweiteren befinden sich auf dem *Carrier-Board* zwei Gigabit-Ethernet Buchsen, welche beide genutzt werden. Über eine der beiden ist die SGM-Box mit dem Sensorkopf verbunden. Die andere Buchse kann

---

[26] Eine vollständige Übersicht der Anschlüsse ist im Benutzerhandbuch des Carrier-Boards unter “Block Diagram” (7 S. 8) zu finden.
[27] Vgl. “Input Power” (7 S. 12)
[28] Vgl. “Selection between mSATA and miniPCIe is done on the MULTI-JUMPER block” (7 S. 22)

genutzt werden, um der SGM-Box Zugang zum Intranet des DLR bereitzustellen. Dies ist insbesondere für den Zugriff auf die Server des Versionskontrollsystems erforderlich.

Ich habe die Software der SGM-Box direkt auf dem Zielsystem entwickelt. Daher habe ich zwei der USB-Ports genutzt, um Maus und Tastatur anzuschließen. Die Bildschirmausgabe erfolgt über den DisplayPort[29] des *Carrier-Boards*. Zur Installation der Betriebssysteme habe ich ein externes, über USB angeschlossenes DVD-Laufwerk verwendet.

Die anderen Anschlüsse des *Carrier-Boards* bleiben ungenutzt. Eine detailliertere Beschreibung technischer Einzelheiten kann den Benutzerhandbüchern beziehungsweise Datenblättern der oben angegebenen Komponenten entnommen werden.

4.3.1.2 Installation des Software-Basissystems

Im Folgenden Abschnitt stelle ich den Prozess der Installation des Basissystems dar. Dies beinhaltet die erforderlichen BIOS-Einstellungen, das verwendete Betriebssystem und die für die Entwicklung genutzten Werkzeuge.

Die erste Herausforderung bei der Inbetriebnahme des eingebetteten Computersystems war, dass ich über keinen der Monitoranschlüsse (DisplayPort, HDMI, VGA) ein Bildsignal bekommen konnte. Ich habe die Hauptplatine daraufhin auf ein anderes *Carrier-Board*[30] montiert, welches alle Monitoranschlüsse herausführt, für den mobilen Betrieb jedoch aufgrund seiner zu großen Abmessungen nicht geeignet ist. An diesem Board erhielt ich über einen der DisplayPorts ein Monitorbild. Ich habe daraufhin die BIOS-Einstellungen so angepasst, dass ich auch

[29] Da die mir zur Verfügung stehenden Monitore lediglich über VGA- und DVI-Anschlüsse verfügen, sind entsprechende Adapter dazwischengeschaltet. Das DisplayPort Signal wird auf HDMI und dies anschließend auf DVI gewandelt.
[30] Es handelt sich hierbei um das conga-TEVAL COM Express Type 6 Evaluation Carrier Board von Congatec

bei Verwendung des in der SGM-Box eingesetzten, kleinformatigen *Carrier-Boards* ein Bildsignal erhalte.

In der Spezifikation der Linux-Treiber für die FPGA-Karte wird als Laufzeitumgebung die Linux-Distribution *SUSE Linux Enterprise Desktop 11 (SLED 11)* mit der *Linux Kernel Version 2.6.32* angeben. Diese Distribution hat sich als Entwicklungssystem für diese Arbeit jedoch aus folgenden Gründen als ungeeignet erwiesen. Als offizielle Quelle für Softwarepakete können nur die Installations-DVDs genutzt werden, welche eine stark eingeschränkte Auswahl bieten. Beispielsweise ist die Versionskontrollsoftware *Subversion (SVN)*, welche am DLR eingesetzt wird, nicht enthalten. Ebenso fehlt eine aktuelle Version der integrierten Entwicklungsumgebung (IDE) *Eclipse*. Teilweise ist es möglich, Softwarepakete für *SLED 11* aus den Quellen von *openSUSE* zu beziehen. Diese Auswahl ist jedoch nicht vollständig und hat in mehreren Fällen zu Kompatibilitätsproblemen geführt.

Dies führte zur der Entscheidung, das Betriebssystem zu wechseln. Ich habe die mir gut vertraute Distribution Ubuntu gewählt. Da das eingebettete System keine eigenständige Grafikkarte besitzt und um nicht unnötig Ressourcen zu verbrauchen, nutze ich die Variante LUbuntu 14.04.1 LTS[31]. Sie ist, abgesehen von der standardmäßig installierten, ressourcenschonenderen Desktopumgebung, identisch zu Ubuntu. Beim installierten Linux Kernel handelt es sich um die Version 3.13.0-39 in der 64Bit-Variante.

4.3.2 Die FPGA-Karte zur Berechnung von SGM

4.3.2.1 Einsatzgebiet der FPGA-Karte

Zur Berechnung des Disparitätsbildes zu einem gegebenen Stereobild verwende ich einen FPGA. Dieser wurde von einem externen Dienstleis-

[31] LTS ist eine Abkürzung für Long Term Support. Das bedeutet dass, die Bereitstellung von Updates für das System über einen langen Zeitraum gewährleistet ist.

ter programmiert und dem DLR in Adlershof zur Verfügung gestellt. Er enthält eine, auf den Einsatz im Straßenverkehr optimierte, Implementierung des SGM Algorithmus. Die Konfiguration der Karte und der Austausch der Bilddaten sind über die PCI Express (PCIe) Schnittstelle implementiert. Bei der verwendeten Hardware handelt es sich um ein Spartan-6 LX75T Development Kit vom Hersteller Avnet.

4.3.2.2 Integration der Hardware

Wie in Abschnitt 4.3.1.1 beschrieben, wird die FPGA-Karte per PCIe angesteuert. Das Computersystem, in welches sie integriert wurde bietet jedoch keine PCIe-Steckplätze an. Daher verwende ich einen Mini PCI Express (mPCIe) zu PCIe-Riser-Adapter[32], um die Karte anzuschließen.

Die FPGA-Karte benötigt zum Betrieb eine Versorgungsspannung von +12 Volt[33]. Diese kann entweder extern oder über die PCIe-Schnittstelle eingespeist werden. Da mPCIe laut Spezifikation[34] keine +12 Volt, sondern nur +1,5 Volt und +3,3 Volt Versorgungsspannung vorsieht, muss diese separat vom Stromanschluss des *Carrier-Boards* auf den „4 Pin Floppy Stromanschluss“ (9) des Riser-Adapters geführt werden.

Um das *Carrier-Board* und die FPGA-Karte mechanisch stabil zu verbinden, habe ich eine Aluminiumplatte entworfen, an welcher beides verschraubt wurde. Dieses Bauteil wurde von einem externen Dienstleister gefertigt.

[32] Die Produktbezeichnung des verwendeten Riser-Adapters lautet “Delock Riser Karte Mini PCI Express > PCI Express x1 links gerichtet 13 cm” (8)

[33] “supply voltages derived from the PCI Express slot or an external 12V Supply” (20 S. 5)

[34] “PCI Express Mini Card provides two power sources: one at 3.3Vaux (3.3Vaux) and one at 1.5V (+1.5V).” (13 S. 43)

5 Die Implementierung der Software

5.1 Die grundlegenden Verarbeitungsschritte

5.1.1 Das Empfangen und Senden der Feederdaten

Zur Anbindung des Systems an das Feedernetzwerk habe ich einen eigenen Feeder implementiert. Diesen bezeichne ich im Folgenden als *SGMOPFeeder*. Er verfügt über einen Eingang, an dem er Stereobilder entgegennimmt, die beispielsweise von einem *StereoFeeder* bereitgestellt werden. An einem Ausgang stellt er nach Durchführung des SGM die entsprechenden Disparitätsbilder zur Verfügung.

Das Empfangen der Stereobilder am Eingang des Feeders und ihre Netzwerkübertragung an die SGM-Box habe ich in einem separaten Thread implementiert. Das Gleiche gilt für den Empfang der, von der SGM-Box über das Netzwerk gesendeten, Disparitäts- und rektifizierten Kamerabilder. Dadurch laufen diese Verarbeitungsschritte losgelöst voneinander und von der Hauptschleife des Feeders ab. Diese befindet sich in der *onRun()*-Methode des *SGMOPFeeders*. Sie stellt die Disparitätsbilder, die der andere Thread empfangen hat, am Ausgang des Feeders zur Verfügung und visualisiert sie in der *IPSApp*. Die Entkopplung der Datenverarbeitung in diesen drei Threads hat den Effekt, dass sowohl in den Sende-, als auch in den Empfangspfaden, mehrere Bilder zwischengespeichert werden können. Das System wird somit durch eine Verzögerung an einer einzelnen Stelle nicht als Ganzes ausgebremst. Beispielsweise kann, während ein Thread auf neue Stereobilder wartet, ein anderer Thread weiterhin Disparitätsbilder empfangen.

Die Kommunikation mit der SGM-Box habe ich in die Klasse *SGMOP-Communicator* ausgelagert. Sie umfasst, neben Funktionen zum Verbindungsaufbau mit dem entfernten System, die Implementierung des Netzwerkprotokolls seitens des Feedernetzwerkes.

Die Konfiguration des *SGMOPFeeders* ist in der Klasse *SGMOPConfiguration* festgelegt. Sie kann durch eine XML-Datei beschrieben werden, welche beim Laden des Feeders gelesen wird. In Abschnitt 5.5 Abbildung 12 habe ich ein Beispiel für eine solche Konfigurationsdatei angegeben.

5.1.2 Die Rektifizierung der Stereobilder

Sobald das erste Stereobild im *SGMOPFeeder* ankommt, berechne ich eine Look-Up-Tabelle zur Rektifizierung der Bilddaten und übertrage sie an die SGM-Box. Da ich annehme, dass sich die Kamerageometrie und die Auflösung der Bilder während eines Laufes nicht ändern, wird sie für die gesamte Bildfolge einer Verbindung verwendet. Ich nutze die Rektifizierungsabbildung, um jedem Punkt im rektifizierten Bild einen Wert zuzuordnen, den ich aus dem unrektifizierten Bild ermittle.

Die Look-Up-Tabelle ist eine m-n-Matrix, wobei *m* die Höhe des Eingangsbildes und *n* seine Breite angibt. In den Einträgen der Matrix werden die Koordinaten des entsprechenden Pixels im unrektifizierten Bild als Tupel von Fließkommazahlen des Typs *double* gespeichert. Wenn beispielsweise der Pixel an der Stelle u=200 v=300 im unrektifizierten Bild auf den Pixel an Stelle u=190 v=290 im rektifizierten Bild abgebildet werden soll, ist dies dadurch angegeben, dass in der Look-Up-tabelle der Matrixeintrag (190,290) die Werte (200.0, 300.0) enthält. Bei der Implementierung des Quelltextes zum Erstellen der Look-Up-Tabelle habe ich mich an der Methode *ImageTransformation.rectifyImage()* aus der *OSVisionLib* des DLR orientiert.

Die für diese Berechnung notwendigen Informationen über die Kamerageometrie hole ich mir aus dem Feedernetzwerk. Sie liegen als statische Parameter des vorgeschalteten *StereoFeeders* vor und werden über den Eingang des Feeders mittels der Methode *getSourceFeederStaticParams(„geometry“)* abgerufen.

Sobald die linke und rechte Look-Up-Tabelle fertig berechnet ist, übertrage ich sie über das Netzwerk an die SGM-Box. Dort werden die beiden hinterlegt und eingehende Stereobilder mit den in ihr gespeicherten Abbildungsvorschriften rektifiziert. Dazu verwende ich den in der *OSLib* implementierten *ImageSubPixelReader*, welcher Methoden zur Interpolation von Bildern bereitstellt. Da hierbei für jeden Pixel des Bildes seine Umgebung betrachtet werden muss, ist dieses Verfahren auf einer CPU relativ zeitaufwändig. Daher habe ich mich auf bilineare Interpolation beschränkt, die im Vergleich zur bikubischen Interpolation zwar weniger genaue Ergebnisse liefert, jedoch schneller zu berechnen ist.

5.1.3 Die Skalierung der Bilder

Damit die rektifizierten Stereobilder von der FPGA-Karte verarbeitet werden können, müssen sie umformatiert werden. Die Eingangsbilder benötigen eine Breite von 1024 Pixeln und eine Höhe von 508 Pixeln. Der in Hardware implementierte Algorithmus ist auf diese Bildgröße optimiert. Die berechneten Dispariätsbilder haben mit einer Breite von 504 Pixeln und einer Höhe von 248 Pixeln in etwa die halbe Auflösung der Eingangsbilder, wobei sich die darin gespeicherten Disparitätswerte auf die Eingangsbilder beziehen. Da diese Bildgrößen in der Regel nicht dem Format der Eingangsbilder entsprechen, die vom Sensorkopf übermittelt werden, müssen diese skaliert werden. Damit können sowohl größere, als auch kleinere Bilder verarbeitet werden. Die horizontale und vertikale Skalierung sind dabei voneinander unabhängig. Zum besseren Verständnis habe ich die einzelnen Skalierungsschritte in Abbildung 4 dargestellt.

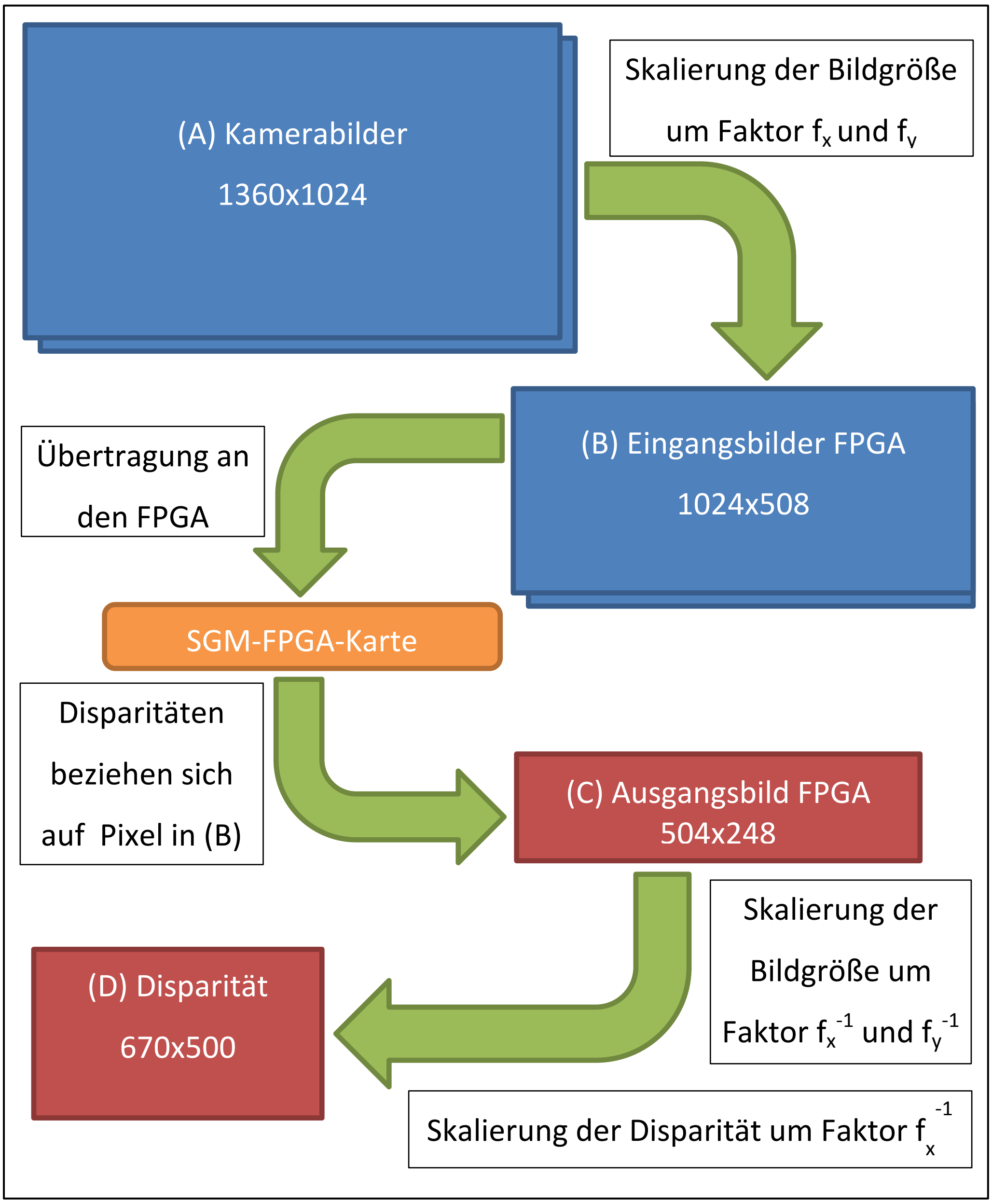


Abbildung 4 - Skalierung der Ein- und Ausgangsbilder

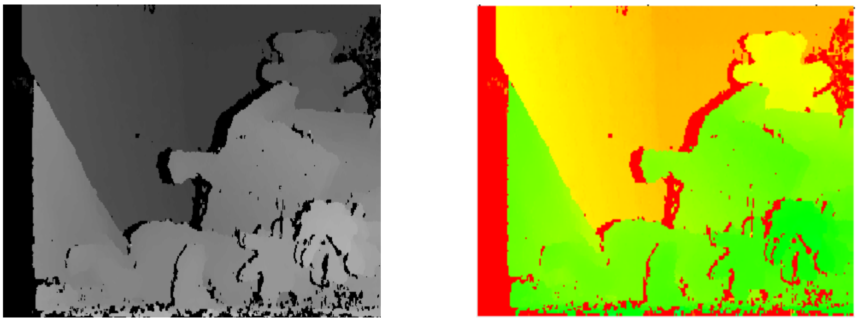
Die Skalierung erfolgt, wie bei der Rektifizierung der Stereobilder, mittels bilinearer Interpolation. Da sich die Veränderung der Größe der Eingangsbilder linear auf das berechnete Disparitätsbild auswirkt, müssen die beiden Skalierungsfaktoren gespeichert und anschließend wieder herausgerechnet werden. Das Disparitätsbild wird also nicht nur in seiner Größe skaliert, sondern, mit dem horizontalen Skalierungsfaktor, auch in seinen Werten angepasst.

5.1.4 Die Visualisierung der Disparitätsbilder

Neben der Bereitstellung der Disparitätsbilder am Ausgang des *SGMOPFeeders* und ihrer optionalen Speicherung im Dateisystem, werden sie in der Benutzeroberfläche der *IPSApp* dargestellt. Da die direkte Darstellung der Disparitäten als Grauwerte ein sehr dunkles Bild ergibt, in dem man wenig erkennen kann, formatiere ich die Bilder zur Visualisierung um. Eine sehr einfache Möglichkeit dafür ist, die Disparitätswerte binär nach links zu schieben, um das Bild aufzuhellen. Alternativ dazu bietet es sich an, die verschiedenen Disparitäten mit unterschiedlichen Farben darzustellen.

Hierzu nutze ich den HSV-Farbraum. Im Gegensatz zum RGB-Farbraum, in dem Farben als Kombination ihrer Rot-, Grün- und Blauanteile definiert sind, gibt man in diesem die Farben als Kombination „des Farbwerts (englisch *hue*), der Farbsättigung (*saturation*) und des Hellwerts (oder der Dunkelstufe) (*value*)“ (10) an. Ich bilde dabei die Disparität ausschließlich auf den Farbwert ab. Um gut erkennbare Farben zu erhalten, setze ich die Sättigung und den Hellwert auf den maximal möglichen Wert. Zur Darstellung der gefärbten Dispariätsbilder in der *IPSApp* wandle ich sie vom HSV- in den RGB-Farbraum um. Die Implementierung ist dabei angelehnt an „Umrechnung HSV in RGB“ (10) auf Wikipedia.

Diese Einfärbung habe ich in Abbildung 5 am Beispiel Middlebury-Datensatzes „Teddy“ (11) demonstriert. Rot entspricht hierbei einer Disparität von Null, beziehungsweise einer Position für die der SGM-Algorithmus keinen gültigen Disparitätswert liefern konnte. Je höher die Disparität, desto mehr geht die Darstellungsfarbe erst ins Gelbe, dann ins Grün und später ins Blaue über.



Graustufen (Disparitätsbild FPGA)

Farbliche Visualisierung (Disparitätsbild FPGA)

Abbildung 5 - Farbliche Visualisierung der Disparitätsbilder

5.2 Die Ansteuerung der FPGA-Karte

5.2.1 Inbetriebnahme und Test der FPGA-Karte

Um die FPGA-Karte unter dem Betriebssystem Linux nutzen zu können, wird ein spezieller Treiber benötigt, welcher von dem externen Dienstleister entwickelt wurde, der auch den FPGA programmiert hat. Nach geringfügiger Korrektur der Include-Direktiven in einem Headerfile des Treibers, konnte ich ihn mithilfe des beiliegenden Makefiles übersetzen.

Das Laden und Entladen des Kerneltreibers erfolgt über die Scripts *s6_pcie_load.sh und s6_pcie_unload.sh*, welche das Unternehmen ebenfalls zur Verfügung gestellt hat. Bevor die Karte verwendet werden kann, muss der entsprechende Treiber geladen werden. Dies erfordert root-Rechte[35] und geschieht über das Script *s6_pcie_load.sh* . Wie in Abbildung 6 dargestellt, sind nach dem Laden des Treibers sowohl das Kernelmodul, als auch die entsprechende Gerätedatei */dev/s6_pcie* vorhanden. Im Kernellog wird angezeigt, dass die Hardware korrekt erkannt wurde und welche Version des Bitstreams aktuell geladen ist.

[35] Administratorrechte

```
root@sgm-linux:/home/adminos/driver# lsmod | grep s6_pcie
root@sgm-linux:/home/adminos/driver# ls /dev/s6*
ls: cannot access /dev/s6*: No such file or directory
root@sgm-linux:/home/adminos/driver# ./s6_pcie_load.sh
root@sgm-linux:/home/adminos/driver# ls /dev/s6*
/dev/s6_pcie
root@sgm-linux:/home/adminos/driver# lsmod | grep s6_pcie
s6_pcie                13366  0
root@sgm-linux:/home/adminos/driver# dmesg | tail
[ 5935.146758] s6_pcie Done unloading module.
[ 6361.773305] s6_pcie Loading module.
[ 6361.773480] s6_pcie PCI BAR0: physical address: 0xf6800000,
size: 0x100000
[ 6361.773869] s6_pcie FPGA version:
[ 6361.773869]   Bitstream:  3.0
[ 6361.773869]   GIT Hash:   c28f7465b320da5d
[ 6361.773869]   Date:       20140415
[ 6361.773869]   Time:       160140
[ 6361.773869]   Algorithms: SGM / EXT Test
[ 6361.775712] s6_pcie Done loading module.
root@sgm-linux:/home/adminos/driver#
```

Abbildung 6 - Laden des Kernelmoduls

Analog dazu führt das Entladen des Treibers, wie in Abbildung 7 dargestellt, dazu, dass das Kernelmodul nicht länger angezeigt wird, die Gerätedatei verschwindet und eine entsprechende Nachricht im Kernellog ausgegeben wird. Sollte sich die Hardware aufgrund eines Fehlers nicht mehr ansteuern lassen, hilft es in der Regel, den Treiber zu entladen und erneut zu laden.

```
root@sgm-linux:/home/adminos/driver# ./s6_pcie_unload.sh
root@sgm-linux:/home/adminos/driver# lsmod | grep s6_pcie
root@sgm-linux:/home/adminos/driver# ls /dev/s6*
ls: cannot access /dev/s6*: No such file or directory
root@sgm-linux:/home/adminos/driver# dmesg | tail
[  6361.773480]  s6_pcie  PCI  BAR0:  physical  address:  0xf6800000,
size: 0x100000
[ 6361.773869] s6_pcie FPGA version:
[ 6361.773869]   Bitstream:  3.0
[ 6361.773869]   GIT Hash:   c28f7465b320da5d
[ 6361.773869]   Date:       20140415
[ 6361.773869]   Time:       160140
[ 6361.773869]   Algorithms: SGM / EXT Test
[ 6361.775712] s6_pcie Done loading module.
[ 6396.937810] s6_pcie Unloading module
[ 6396.938065] s6_pcie Done unloading module.
root@sgm-linux:/home/adminos/driver#
```

Abbildung 7 - Entladen des Kernelmoduls

Mit dem Treiber der FPGA-Karte habe ich fertige Testfälle zur Verfügung gestellt bekommen. Mithilfe dieser habe ich sichergestellt, dass die Hardware korrekt funktioniert. Darin sende ich Stereobilder an die FPGA-Karte, empfange die entsprechenden Dispariätsbilder und vergleiche sie mit vorliegenden Referenzbildern. Um die Ergebnisse der FPGA-Karte auch noch optisch zu validieren, habe ich die Testfälle so abgeändert, dass die empfangenen Dispariätsbilder unter Verwendung entsprechender, schon enthaltener Funktionen gespeichert werden. Dadurch und durch ihre Spezifikation habe ich festgestellt, dass die FPGA-Karte für Bildbereiche, denen sie keine Disparitätswerte zuordnen kann, die Disparität Null angibt. Das bedeutet, dass man mit der FPGA-Karte nicht zwischen Bereichen unterscheiden kann, die vom SGM-Algorithmus nicht erkannt wurden, und solchen, die im Objektraum sehr weit entfernt liegen und somit im linken und rechten Bild mit den gleichen Bildkoordinaten abgebildet werden.

5.2.2 Ansteuerung mittels Userspace-Library

Zur Ansteuerung der FPGA-Karte existiert eine in C programmierte Bibliothek. Im Folgenden beschreibe ich, wie ich diese Bibliothek nutze.

Zuerst stelle ich eine Verbindung zur FPGA-Karte her. Dies erfolgt mit der Funktion *pcie_sgm_open*, welche bei erfolgreicher Ausführung einen Handler zurückgibt, mit dem die Hardware angesprochen werden kann. Dieser dient allen weiteren Funktionen als Referenz für die Hardwareschnittstelle. Falls das Öffnen fehlschlägt, wird *NULL* zurückgegeben. Mit der Funktion *pcie_sgm_close* kann diese Verbindung wieder gelöst werden.

Wenn die Verbindung zur FPGA-Karte hergestellt wurde, setze ich mit der Funktion *pcie_sgm_set_parameters* die Parameter für den SGM-Algorithmus. Desweiteren lese ich mit der Funktion *pcie_sgm_get_image_resolution* aus, welche Auflösungen die Eingangs- und Ausgangsbilder haben müssen und allokiere den entsprechenden Speicher.

Sobald ich das erste Stereobild empfangen, rektifiziert, auf die zuvor ausgelesene Auflösung skaliert und in das von der FPGA-Karte erwartete Format konvertiert habe, lade ich es mittels der Funktion *pcie_sgm_transfer_images_16bit* in ihren Speicher. Dies löst die Berechnung des Disparitätsbildes aus.

Ich signalisiere dies einem weiteren Thread, der daraufhin mit der Funktion *pcie_sgm_get_disparity_images* auf das Ergebnis der Berechnung wartet. Diese Signalisierung beinhaltet die entsprechenden Skalierungsfaktoren in x- und y-Richtung. Sobald ich das Disparitätsbild empfangen habe, formatiere und skaliere ich es mit den Kehrwerten der übermittelten Faktoren wieder entsprechend zurück. Damit ist die durch die FPGA-Karte durchzuführende Berechnung abgeschlossen und das Resultat kann über das Netzwerk an den Sensorkopf übertragen werden.

5.3 Die Kommunikation der einzelnen physischen Komponenten

5.3.1 Die Anforderungen an das eingesetzte Netzwerkprotokoll

Wie oben beschrieben, sind der Sensorkopf und die SGM-Box über Ethernet verbunden. Um Daten untereinander austauschen zu können, müssen die beiden Rechnersysteme ein gemeinsames Netzwerkprotokoll implementieren. Es existiert eine Vielzahl an Protokollen und die Wahl eines geeigneten Kandidaten hängt von mehreren Faktoren ab. Eigenschaften, auf die ich bei der Auswahl einer geeigneten Methode besonderen Wert gelegt habe, sind Geschwindigkeit, Wartbarkeit, Einfachheit und Portierbarkeit.

Die in beide Richtungen zu übertragenden Bilder und vor allem die Look-Up-Tabelle stellen relativ große Datenmengen dar. Daher muss das verwendete Protokoll zulassen, dass die Daten schnell verarbeitet werden können. Außerdem soll der Overhead, also die durch das verwendete Protokoll zu den Nutzdaten hinzugefügte Datenmenge, möglichst gering sein.

Da die SGM-Box und der entsprechende Feeder auch nach meiner Masterarbeit noch am DLR eingesetzt werden sollen, muss das von mir entwickelte System gut wartbar sein. Dazu gehört auch, dass das Netzwerkprotokoll im Nachhinein an sich möglicherweise verändernde Anforderungen angepasst und um neue Funktionen erweitert werden kann. Dies soll ohne nennenswerte Eingriffe in die zu dem entsprechenden Zeitpunkt implementierten und getesteten Funktionen möglich sein.

Es gibt mehrere Gründe, warum das implementierte Protokoll möglichst schlicht und einfach sein sollte. Zum einen wird dadurch die Fehler-

wahrscheinlichkeit minimiert. Wenn der zur Interpretation der empfangenen Daten erforderliche Parser klein gehalten werden kann, ist er auch robuster gegenüber Programmierfehlern, als ein vergleichsweise großer Parser für ein komplexes Protokoll. Zum anderen wird Dritten durch ein leicht verständliches Protokoll die Einarbeitung in das vor mir entwickelte System erleichtert.

Da die Software der SGM-Box unter Linux und das Sensorframework im Sensorkopf unter Windows läuft, muss das Protokoll ohne großen Aufwand auf beiden Betriebssystemen implementiert werden können. Portierbarkeit ist also ein weiteres Kriterium für die Wahl des Protokolls, mit dem die beiden Systeme kommunizieren sollen.

5.3.2 Eine Betrachtung verschiedener Netzwerkprotokolle

Ich beschränke mich hier auf die Betrachtung dreier möglicher Protokolle, auf welche ich die Auswahl mittels Internetrecherche und in der Diskussion mit meinen Kollegen reduziert habe. Zuerst betrachte ich die Kombination von *Web Services Description Language (WSDL)* und *Simple Object Access Protocol (SOAP)*, welche auch in einem anderen Projekt in der Arbeitsgruppe, in der ich meine Abschlussarbeit schreibe, eingesetzt wurde. Als nächstes gehe ich auf das *Message Passing Interface (MPI)* ein und betrachte anschließend die Option, ein eigenes auf das Problem angepasstes Protokoll zu implementieren.

WSDL ist eine auf XML basierende Beschreibungssprache, mittels derer sich die Schnittstelle zu einem Service definieren lässt. Die wesentlichen Elemente der Sprache sind *Services*, *Types*, *Messages*, *Operations* und *Ports*. Ein *Service* stellt dabei eine Sammlung von Endpunkten im Netzwerk dar[36]. Die einzelnen *Services* verfügen über *Ports*, welche *Operations* zur Verfügung stellen. Dadurch werden *Messages* ausgetauscht. Die in ihnen enthaltenen Datentypen werden in *Types* definiert.

[36] Vgl. „A WSDL document defines services as collections of network endpoints" (15)

In WSDL sind vier Arten von *Ports* (*Port Types*) vorgesehen. *One-way Ports* können *Messages* lediglich empfangen, aber nicht darauf antworten[37]. *Notification Ports*, im Gegensatz dazu, senden *Messages* ausschließlich, können aber keine empfangen[38]. *Request-response Ports* warten auf eine *Message* und senden eine entsprechende Antwort[39]. *Solicit-response Ports* arbeiten umgekehrt. Sie senden zuerst eine *Message* und empfangen anschließend eine entsprechende Antwort[40].

Bei der Kommunikation des Sensorkopfes mit der SGM-Box werden Daten bidirektional ausgetauscht. Daher beschränkt sich die Auswahl an *Port Types* auf den *Request-response Port* den *Solicit-response Port*. Wenn der Service auf der SGM-Box unter Linux laufen sollte, würde ich dort einen *Request-response Port* definieren. Dieser wartet auf das Stereobild und antwortet mit dem Tiefenbild. Trotz längerer Suche im Internet konnte ich für WSDL-Funktionalität unter Linux lediglich Client-Bibliotheken, wie beispielsweise das Sourceforge-Projekt *WSDLPull*, finden. Daher muss der Service im Sensorkopf unter Windows ausgeführt und über einen *Solicit-response Port* zur Verfügung gestellt werden. Von der Verwendung dieses *Port Types* wird jedoch aus Gründen der Interoperabilität abgeraten[41].

Ein weiterer Entscheidungsgrund gegen die WSDL ist, dass sie von den Bibliotheken, die ich gefunden habe, stets mit SOAP eingesetzt wird. SOAP ist ein auf XML basiertes Protokoll, welches für den strukturierten Datenaustausch vorgesehen ist. Die Übertragung roher Binärdaten wird jedoch nicht unterstützt. Diese müssen durch ein Verfahren, wie Base64 so kodiert werden, dass nur druckbare Zeichen enthalten sind. Die

---

[37] Vgl. „One-way. The endpoint receives a message." (15)
[38] Vgl. „Notification. The endpoint sends a message." (15)
[39] Vgl. „Request-response. The endpoint receives a message, and sends a correlated message." (15)
[40] Vgl. "Solicit-response. The endpoint sends a message, and receives a correlated message." (15)
[41] Vgl. "R2303 [-] A DESCRIPTION MUST NOT use Solicit-Response and Notification type operations in a wsdl:portType definition." (23)

Konvertierung der Daten kostet dabei zusätzliche Rechenzeit und das auf 64 Zeichen reduzierte Alphabet erzeugt eine steigende Netzwerklast. Ein Byte kann $2^8 = 256$ verschiedene Werte annehmen, ein Zeichen in Base64-Kodierung lediglich $2^6 = 64$ verschiedene Werte. Die Bilddaten werden zur Übertragung also auf $^8/_6 = {}^4/_3$ ihres ursprünglichen Speicherplatzbedarfs vergrößert.

Obwohl WSDL die Implementierung eines sehr gut wartbaren und leicht verständlichen Protokolls ermöglicht, haben die nur bedingt gegebene Plattformunabhängigkeit und die Leistungseinbußen bei der Datenübertragung dazu geführt, dass ich mich gegen dessen Verwendung entschieden habe.

Als Alternative habe ich das Message-Passing Interface (MPI) untersucht, welches dem Datenaustausch in einem parallelisierten Rechnersystem dient. Es kann sowohl für Punkt-zu-Punkt Verbindungen, als auch für Kommunikationsgruppen eingesetzt werden[42]. Vor allem für letztere sieht MPI sehr viele Funktionen vor, die ich jedoch nicht nutze, da ich lediglich eine Punkt-zu-Punkt Verbindung zwischen dem Sensorkopf und der SGM-Box benötige. Eines der Ziele bei der Entwicklung von MPI war hohe Leistungsfähigkeit[43]. Es ermöglicht den effizienten Austausch von Binärdaten. Als Netzwerkprotokoll nutzen die meisten Implementierungen TCP[44] (Transmission Control Protocol).

Im Gegensatz zu WSDL erlaubt MPI keine Beschreibung der Struktur der zu übertragenden Daten. MPI bietet in dem von mir gegebenen Anwendungsfall also kaum Mehrwert gegenüber der reinen Nutzung von TCP-Sockets. Ich habe daher von der Verwendung von high-level Protokollen abgesehen und ein eigenes Protokoll auf Basis von TCP implementiert.

---

[42] Vgl. „Both point-to-point and collective communication are supported" (18)
[43] Vgl. "MPI's goals are high performance, scalability, and portability." (18)
[44] Vgl. „Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers, with sockets and Transmission Control Protocol (TCP) used in the transport layer." (18)

5.3.3 Die detaillierte Beschreibung des implementierten Netzwerkprotokolls

Um zwischen der SGM-Box und dem Sensorkopf Daten, wie die Kamerabilder, das Tiefenbild und die Look-Up-Tabelle für die Bildrektifizierung zu übertragen, habe ich ein eigenes Protokoll entwickelt. Dabei habe ich neben guter Erweiterbarkeit und effizientem Datenaustausch besonderen Wert auf Einfachheit und leichte Verständlichkeit gelegt.

Ich habe den Datenaustausch mittels TCP-Sockets implementiert. Da TCP garantiert, dass die zu übertragenden Daten entweder korrekt zugestellt werden oder ein entsprechender Fehler erkannt wird, muss ich keine zusätzlichen Mechanismen zur Fehlererkennung, wie beispielsweise Prüfsummen oder paketweise Empfangsbestätigungen, vorsehen. Die SGM-Box übernimmt dabei die Rolle des Servers und der Sensorkopf die des Clients. Das bedeutet, dass die SGM-Box auf einem TCP-Port eingehende Verbindungen vom Sensorkopf entgegennimmt. Sobald eine Verbindung hergestellt wurde, können Informationen bidirektional ausgetauscht werden.

Beim Datenaustausch zwischen mehreren verschiedenen Rechnern sollte darauf geachtet werden, dass unterschiedliche Architekturen unterschiedliche Speicherdarstellungen von Zahlen, die mehr als ein Byte Speicherplatz einnehmen, besitzen können. Die beiden am meisten verbreiteten Formen sind Big-Endian und Little-Endian. „Big-Endian Architekturen […] speichern das höchstwertige Byte an der niedrigsten Adresse“ (12 S. 856) und „Little-Endian Architekturen […] speichern das niederwertigste Byte an der niedrigsten Adresse“ (12 S. 856). In der Regel werden Daten im Netzwerkverkehr als Big-Endian formatiert. Daher wird diese Darstellung auch *network byte-order* genannt. Die meisten Betriebssysteme stellen Bibliotheksfunktionen zur Verfügung, mit denen eine Zahl zwischen der jeweiligen eigenen Darstellung und

der in *network byte-order* konvertiert werden kann. Für die Bytereihenfolge der Netzwerkübertragung von Fließkommazahlen existiert meines Wissens keine einheitliche Vorschrift.

Die einzelnen Übertragungen habe ich dabei in Pakete gekapselt, die ich anhand eines vorangestellten Pakettypen unterscheide. Dieser ist als *uint8*[45] implementiert, was eine Erweiterbarkeit auf bis zu 2^8=256 verschiedene Pakettypen ermöglicht. Die verschiedenen Pakettypen sind in Tabelle 1 dargestellt. Dabei ist angegeben, wer den entsprechenden Pakettyp in meiner Implementierung sendet und wer ihn empfängt. Bis auf den Spezialfall, dass ein Paket als ungültig markiert ist, sind Client und Server bei allen Pakettypen entweder Sender oder Empfänger.

Pakettyp	Funktion	Sender	Empfänger
0	Stereobild	SGM-Box	Sensorkopf
1	SGM Parameter	Sensorkopf	SGM-Box
2	Look-Up-Tabelle	Sensorkopf	SGM-Box
3	Disparitätsbild	SGM-Box	Sensorkopf
4	Rektifiziertes Bild	SGM-Box	Sensorkopf
5	Verschiedene Optionen	Sensorkopf	SGM-Box
255	Ungültiges Paket	beide	beide

Tabelle 1 - Pakettypen

Nach dem Pakettyp werden die spezifischen Daten übermittelt. Im Falle eines Stereobildes wird zuerst das linke, dann das rechte Bild übertragen. Jedem der beiden Bilder wird seine jeweilige Breite und Höhe als *uint16* vorangestellt. Da nur Bilder mit 8 Bit Grauwerttiefe übertragen werden, ergibt sich die Größe des zu übermittelnden Bildes in Bytes als Produkt der beiden Werte. Sobald die entsprechende Datenmenge empfangen wurde, wird auf den nächsten Pakettypen gewartet. Dispari-

[45] Die Abkürzung *uint8* bezeichnet den Datentypen einer vorzeichenlosen Ganzzahl mit 8 Bits. Analog dazu sind *uint16*, *uint32* und *uint64* definiert. Für vorzeichenbehaftete Zahlen verwende ich die Bezeichnungen *sint8*, *sint16*, *sint32* und *sint64*.

tätsbilder, rektifizierte Bilder und die Look-Up-Tabellen werden auf analoge weise übertragen.

Das Paket mit den SGM Parametern hat eine feste Größe von 9 Bytes. Diese ergeben sich aus 1 Byte Pakettyp, je 2 Bytes für die als *uint16* kodierten Parameter P_1 und P_2 und je 1 Byte für die 4 Flags *enableGaussFilter*, *enableFeedback*, *enableFastLRCheck* und *enableSkyInit*. Die Übertragungsreihenfolge entspricht der ihrer Nennung in diesem Absatz. Da die Parameter nur einmal zum Beginn einer Verbindung übertragen werden, habe ich darauf verzichtet, die Flags binär kodiert auf ein einzelnes Byte zu komprimieren.

Desweiteren können noch zusätzliche Informationen übertragen werden, die im Projektverlauf als Anforderung hinzugefügt wurden. Diese legen fest, ob das Stereobild auf die Höhe und Breite der FPGA-Karte skaliert oder ob, unter Voraussetzung hinreichender Größe des Eingangbildes, nur ein Ausschnitt in Originalauflösung verwendet werden soll. Zuerst wird das Flag *enableScaling* als 1 Byte übertragen. Dieses gibt an, ob skaliert werden soll oder nicht. Anschließend wird als zwei *uint16* die horizontale und dann vertikale Position des linken oberen Ursprungs eines möglichen Ausschnittes übertragen.

Zur einfacheren Handhabung des Datenaustausches verwende ich die Klassen *BinaryStreamWriter* und *BinaryStreamReader* der *OSLib*. Diese ermöglichen es, primitive Datentypen typsicher in einen Stream zu schreiben beziehungsweise aus ihm zu lesen. Die Übertragung von Werten des Typs *unsigned char* habe ich hierbei angepasst, sodass diese genau wie Daten vom Typ *char* als einzelnes Byte und nicht als 4 Bytes übertragen werden.

Um *BinaryStreamWriter* und *BinaryStreamReader* verwenden zu können, werden *OutputStreams* beziehungsweise *InputStreams* benötigt. Daher habe ich die Klassen *TCPInputStream* und einen *TCPOut-*

putStream implementiert, welche sich davon ableiten und eine stream-basierte Datenübertragung über TCP ermöglichen. Zum Datenaustausch verwenden sie einen *TCPClientSocket*, der bei der Erzeugnung der Streams als Parameter an den Konstruktor übergeben wird. Der Socket wird hierbei als *SmartPointer* gespeichert. Wenn beim Schließen des Streams keine weiteren Referenzen mehr auf den Socket bestehen, wird auch dieser geschlossen. Sollten noch andere Programmteile den Socket verwenden, bleibt er geöffnet. In der übergeordneten Programmebene, also der Ebene, in der die Streams erzeugt wurden, muss der Socket also nicht mehr verwaltet werden.

Zum Test meiner Implementierungen der Klassen *TCPInputStream* und *TCPOutputStream* habe ich den Testfall *Test_TCPStream* implementiert. Dabei werden über einen lokalen TCP-Socket Daten an einen Server gesendet, der diese unverändert wieder zurückschreibt. Falls unerwartete Fehler auftreten oder die vom Client gesendeten Daten nicht denen entsprechen, die er auch wieder empfängt, schlägt der Test fehl. Andernfalls ist er erfolgreich.

5.3.4 Die Beschreibung der Kompression von Disparitätsbildern

Die Dispariätsbilder, die der FPGA liefert, sind im 7.4 Bit Festkommaformat *(engl. fixed-point format)* angegeben. Das bedeutet, dass 7 Bits die Vorkommastelle und 4 Bits die Nachkommastelle angeben. Bei der Nachkommastelle spricht man hierbei von Subpixel-Disparität. Ein Pixel im Disparitätsbild benötigt also 11 Bits Speicherplatz. Aus Gründen der einfacheren Verarbeitbarkeit, wird in der Software des Linuxtreibers für jeden Pixel ein 16-Bit Wert verwendet. Die obersten 5 Bits sind dabei stets gleich Null.

Um die Dispariätsbilder über das Netzwerk zu übermitteln, muss man nur die Bits übertragen, die tatsächlich Informationen enthalten. Die

Übertragung der Dispariätsbilder kann somit beschleunigt werden. Dabei sollte die Kompression der Daten jedoch nicht länger dauern, als die Übertragung der unkomprimierten Daten. Ich habe hierfür ein Kompressionsverfahren entwickelt, dass sich relativ einfach und effizient implementieren lässt.

Ein Pixel wird dabei verlustfrei von 16Bit auf 12Bit Datenvolumen komprimiert. Dabei werden 2 Pixel mit 7.4-Festkommadarstellung in 3 Bytes statt in 4 Bytes übertragen. Die jeweils 4 Bit Subpixelanteil der beiden Pixel werden in einem Byte zusammengefasst. Die 7 Bit Vorkommaanteil eines jeden Pixels werden in je einem einzelnen Byte übertragen, dessen oberstes Bit ungenutzt bleibt. Die Übertragungsreihenfolge ist so gewählt, dass zwischen zwei Bytes mit Vorkommaanteilen das ihnen zuzuordnende Subpixelbyte übermittelt wird. Die niederwertigen 4 Bits enthalten hierbei die Subpixel des zuerst übermittelten (beziehungsweise *geraden*) Pixels, die höherwertigen 4 Bits die des als zweites übermittelten (beziehungsweise *ungeraden*) Pixels. Die Bezeichnungen *gerade* und *ungerade* beiziehen sich hierbei auf die Nummer in der Übertragungsreihenfolge der Pixel.

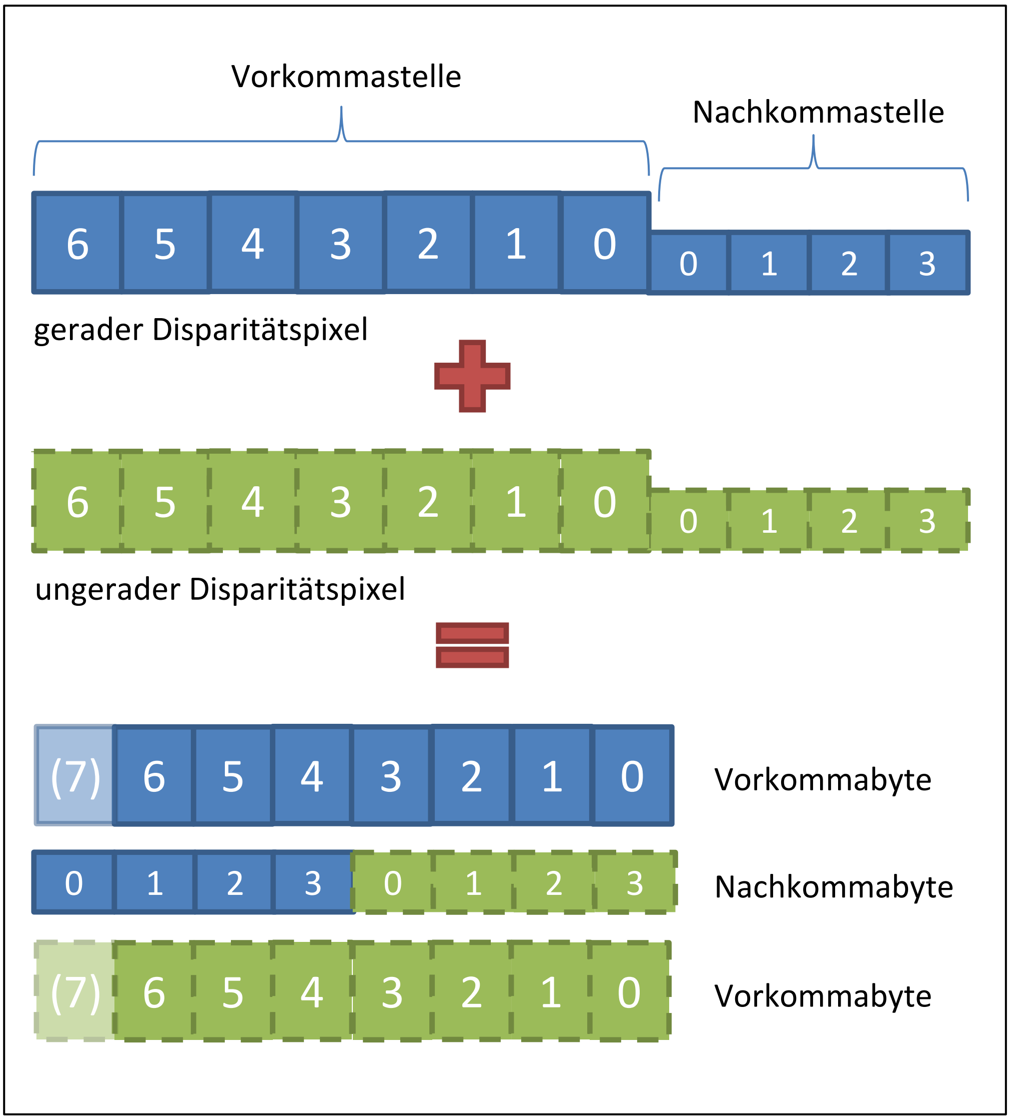


Abbildung 8 - Kompression der Dispariätsbilder

Dieses Verfahren ist in Abbildung 8 dargestellt. Jedes Kästchen repräsentiert ein Bit. Die großen Kästchen sind Vorkommastellen, die kleinen sind Nachkommastellen. Die blauen Bits mit durchgezogenem Rand bilden das zuerst übertragene *(gerade)* Pixel, die grünen Bits mit gestricheltem Rand das als zweites übertragene *(ungerade)* Pixel.

5.4 Die Parallelisierung der Verarbeitungsprozesse

5.4.1 Die Aufteilung in einzelne Threads

Um die verfügbaren Ressourcen besser auszunutzen, unterteile ich den gesamten Verarbeitungsprozess, vom unrektifizierten Stereobild bis hin

zum rektifizierten Disparitätsbild, in einzelne kleinere Unterschritte. Diese Unterschritte sollen keine gemeinsamen Ressourcen nutzen und können somit nebenläufig ausgeführt werden. Beispielsweise ist es möglich, dass während der FPGA das Disparitätsbild berechnet, die CPU bereits das nächste Bild rektifiziert, anstatt darauf zu warten, dass der FPGA fertig ist. Parallel dazu kann, um ein weiteres Beispiel anzuführen, auch schon das nächste Stereobild über das Netzwerk empfangen oder ein Disparitätsbild gesendet werden. Dadurch entsteht eine Pipeline, deren Stufen die einzelnen Verarbeitungsschritte darstellen. Zur Implementierung dieser einzelnen Stufen verwende ich Threads, die über Dateneingänge und -ausgänge verfügen, mittels derer sie mit anderen Stufen der Pipeline kommunizieren können.

In Abbildung 9 sind die einzelnen Komponenten, ihre Verknüpfung und die von ihnen ausgetauschten Informationen dargestellt. Links unten ist die FPGA-Karte zu sehen. Oben in der Mitte ist die Software der SGM-Box mit dem Symbol des Linux-Pinguins dargestellt. Das unter Windows ausgeführte Sensornetzwerk ist mit einem Foto des IPS-Sensorkopfes dargestellt.

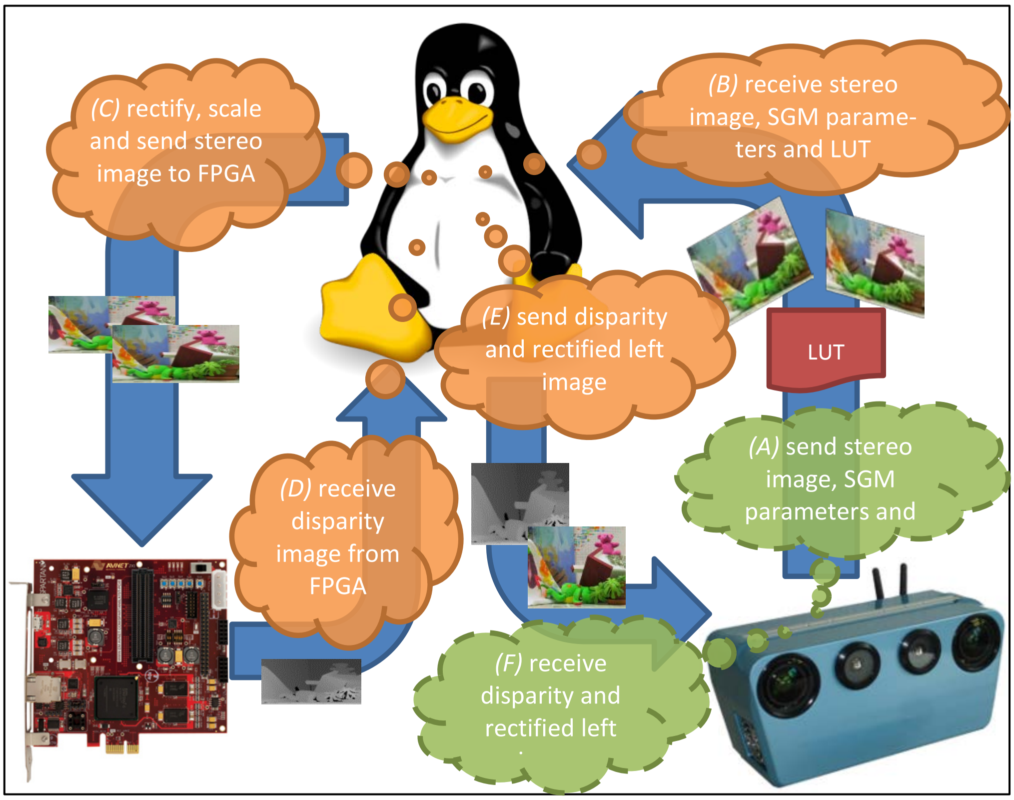


Abbildung 9 - Aufteilung in unabhängige Komponenten [46] [47] [48] [49]

Die verschiedenen Stufen sind als sechs Denkblasen dargestellt, die von *A* bis *F* durchnummeriert sind. Die grünen Denkblasen mit gestrichelten Rändern symbolisieren dabei Threads, die unter Windows im Sensornetzwerk aufgeführt werden; die orangenen mit durchgezogen Rändern jene die auf der SGM-Box unter Linux laufen. Die Kommunikation der Threads untereinander läuft asynchron ab. Das setzt voraus, dass Daten, die ein Thread an einen anderen sendet, zwischengespeichert werden. Der sendende Thread kann weiterlaufen, ohne darauf warten zu müssen, dass der empfangende Thread die Daten abholt.

---

[46] Die in Abbildung 9 enthaltenen Stereo- und Disparitätsbilder sind dem Middlebury-Datensatz „Teddy“ (12) entnommen.
[47] Der in Abbildung 9 verwendete Linux-Pinguin stammt von Wikimedia.
Quelle: http://commons.wikimedia.org/wiki/File%3ATux.svg (31.01.2015)
[48] Die in Abbildung 9 abgebildete FPGA-Karte dem entsprechenden Userguide von Avnet.
Quelle: “Spartan-6 LX75T FPGA Development Board Picture” (20 S. 6)
[49] Die in Abbildung 9 dargestellte IPS-Box stammt von der Website des DLR.
Quelle: http://www.dlr.de/os/en/desktopdefault.aspx/tabid-9967/17040_read-41235/ (31.01.2015)

Eine Iteration des gesamten Prozesses beginnt bei dem nicht-rektifizierten Stereobild und endet bei den rektifizierten Bilddaten und dem dazugehörigen Disparitätsbild. Zuerst nimmt *Thread A* die Stereobilder im Feedernetzwerk entgegen und sendet sie per TCP an die SGM-Box. Beim ersten Stereobild berechnet er außerdem die Look-Up-Tabelle zur Rektifizierung der Bilder und überträgt diese und die Parameter für den SGM-Algorithmus gemeinsam mit dem Stereobild. Dort nimmt der *Thread B* die entsprechenden Daten entgegen und sendet sie an *Thread C*. Dieser rektifiziert damit das Stereobild, überführt es in das von der FPGA-Karte unterstützte Format und lädt es anschließend in ihren Speicher. Zuletzt sendet er, während die FPGA-Karte das Disparitätsbild berechnet, das linke rektifizierte Bild an *Thread E*, der es über das TCP-Netzwerk an den *Thread F* weiterleitet.

Sobald die FPGA-Karte die Berechnung des Disparitätsbildes abgeschlossen und dies mit einem entsprechenden Interrupt an *Thread D* gemeldet hat, formatiert dieser die Daten wieder um und sendet sie an *Thread E*. Dort wird das Disparitätsbild, wie zuvor das linke rektifizierte Bild, an *Thread F* im Sensorkopf übermittelt. Dieser *(Thread F)* stellt schließlich das rektifizierte Bild und das Disparitätsbild grafisch dar und an seinen Ausgängen für den nächsten Feeder zur Verfügung.

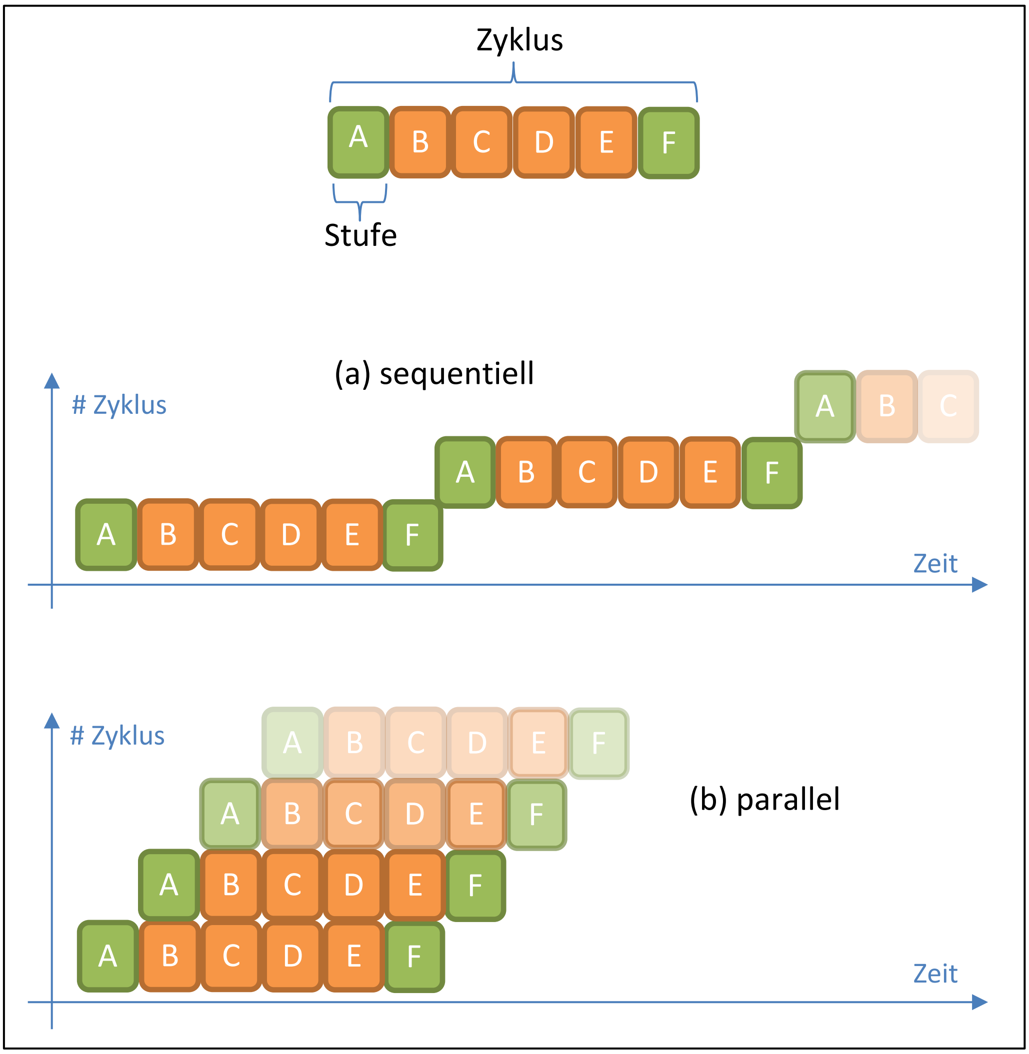


Abbildung 10 - Parallelisierung durch eine Pipeline

Damit ist ein Bearbeitungszyklus abgeschlossen. In einer sequentiellen Implementierung würde jetzt, wie in Abbildung 10a dargestellt, ein neuer Zyklus beginnen. Die Bildrate im Gesamtsystem würde also der inversen Summe der Ausführungszeiten aller Unterschritte entsprechen. Da sich die Verarbeitungszyklen in einer Pipeline jedoch, wie in Abbildung 10b dargestellt überlagern, ist das limitierende Element der langsamste Verarbeitungsschritt. Die Bildrate entspricht also dem Kehrwert der Summe seiner Verarbeitungszeit und einem konstanten nicht-parallelisierbaren Anteil. Dieser kommt dadurch zu Stande, dass die Verarbeitung in einer Pipeline aufwendiger zu realisieren ist, als eine sequentielle Implementierung. Er beinhaltet beispielsweise die Daten-

übergabe zwischen den einzelnen Stufen und die Verwaltung der Threads.

5.4.2 Die rechnerinterne Kommunikation einzelner Threads

5.4.2.1 Der Ringpuffer als verwendete Datenstruktur

Damit die einzelnen Threads innerhalb eines physischen Systems untereinander kommunizieren können, wird eine gemeinsame Datenstruktur benötigt. Ich habe mich dazu entschieden, diese als typ- und threadsichere Ringpuffer zu implementieren.

Bei einem Ringpuffer handelt es sich um eine Warteschlange (engl. Queue) mit fester Kapazität[50], auf der fortlaufend geschrieben und gelesen werden kann, ohne dass bereits enthaltene Daten kopiert oder verändert werden müssen. Da die Daten, die man als erstes hineingibt, auch wieder als erstes ausgelesen werden, spricht man hierbei von einer FIFO (First-In, First-Out). Da Ringpuffer relativ speicher- und laufzeiteffizient sind und sich leicht implementieren lassen, werden sie vor allem in eingebetteten Systemen häufig eingesetzt. Neben den eigentlichen Daten werden lediglich drei Zeiger benötigt. Ein Zeiger markiert den Anfang des genutzten Speicherbereichs, einer die aktuelle Leseposition und einer die aktuelle Schreibposition. Die Lese- und Schreibzugriffe auf den Ringpuffer sind in $O(1)$ durchführbar[51]. Ihre Laufzeit kann, mit anderen Worten, als konstant betrachtet werden und ist folglich unabhängig von sowohl der Kapazität des Puffers, als auch der Anzahl der aktuell darin gespeicherten Elemente. Für den Fall, dass aus einem leeren Ringpuffer gelesen oder in einen vollen Ringpuffer geschrieben werden soll, ist ein entsprechendes Ausnahmeverhalten[52] zu definieren.

---

[50] Vergleiche hierzu “Circular buffering makes a good implementation strategy for a queue that has fixed maximum size.” (16)

[51] “Each operation takes $O(1)$ time.” (14 S. 235)

[52] Vergleiche hierzu “error checking for underflow and overflow” (14 S. 235)

Ich nutze die Implementierung aus der *OSLib*, welche unterschiedliche Funktionen für das jeweilige Ausnahmeverhalten bereitstellt.

Beim Schreiben in den Ringpuffer wird die aktuelle Schreibposition ausgelesen, der zu schreibende Wert an die entsprechende Speicherstelle übertragen und die Schreibposition um Eins inkrementiert. Wenn diese sich dadurch aus dem gültigen Speicherbereich des Ringpuffers herausbewegt, wird sie wieder auf den Anfang desselbigen gesetzt. Anschließend wird die neue Schreibposition im Ringpuffer aktualisiert. Das Lesen läuft analog dazu ab.

Durch die Typsicherheit wird garantiert, dass von dem Ringpuffer nur Daten eines bestimmten Typs gespeichert und geladen werden können. Dies wird zum Zeitpunkt der Übersetzung vom Compiler geprüft und gegebenenfalls mit einem entsprechenden Fehler angegeben. Durch Threadsicherheit können mehrere Threads auf ein und denselben Ringpuffer zugreifen, ohne dass dieser einen ungültigen Zustand annehmen kann. Dazu wird die Datenstruktur um Mechanismen erweitert, die zur Laufzeit aktiv verhindern, dass sich zwei zeitgleich auf sie initiierte Zugriffe überlappen.

Um eine threadsichere Nutzung der Ringpuffer zu ermöglichen, muss garantiert werden, dass nie zwei Threads gleichzeitig auf ihn zugreifen. Wenn beispielsweise zwei Threads im selben Moment beginnen, einen Schreibzugriff durchzuführen, so würden sie beide die aktuelle Schreibposition des Ringpuffers lesen, um ihre Daten an der entsprechenden Speicherposition zu hinterlegen. Da beide dieselbe Position hierfür nutzen würden, wäre nicht definiert, welcher Wert anschließend in der entsprechenden Speicherzelle stünde. Desweiteren würden beide die Schreibposition um Eins erhöhen und in den gemeinsamen Speicher zurückschreiben. Um dies zu vermeiden, habe ich die Klasse *Mutex* aus der *OSLib* genutzt. Jeder Ringpuffer besitzt genau einen *Mutex*. Dieser

kann gesperrt und entsperrt werden. Ist der *Mutex* gesperrt, kann er nicht erneut gesperrt werden. Dadurch, dass ich zu Beginn jeder Methode des Ringpuffers diesen *Mutex* sperre und am Ende wieder entsperre, garantiere ich, dass sich in dem Bereich zwischen diesen beiden Operationen nie mehrere Threads befinden.

Sollte in diesem Bereich jedoch eine *Exception*[53] ausgelöst werden, sodass die Methode nicht bis zum Ende ausgeführt wird, bleibt der Ringpuffer gesperrt. Daher habe ich die manuelle Sperrung und Entsperrung durch die Verwendung der, ebenfalls in der *OSLib* implementierten, Klasse *Lock* ersetzt. Einem *Lock* wird bei der Instanziierung durch den Konstruktor ein Mutex übergeben. Solange das *Lock* gültig ist, bleibt der *Mutex* gesperrt. Erst in seinem Destruktor gibt das *Lock* den *Mutex* wieder frei. Ich erzeuge also am Anfang einer jeden Methode des Ringpuffers eine lokale Variable des Typs *Lock* auf seinem *Mutex*. Sobald die Funktion verlassen wird, also auch im Falle einer *Exception*, verliert das *Lock* seine Gültigkeit und gibt den *Mutex* wieder frei.

Zur Verifizierung des threadsicheren Ringpuffers habe ich einen entsprechenden Testfall implementiert. Er enthält zwei Threads, die sich einen Ringpuffer teilen. Der eine Thread schreibt die Zahlen von 0 bis 100 in den Ringpuffer. Der andere Thread liest die Daten aus dem Ringpuffer aus und vergleicht, ob auch wirklich die Zahlen 0 bis 100 übertragen wurden. Nur wenn alle Werte korrekt übertragen wurden, gilt der Test als bestanden. Die Kapazität des Puffers ist hierbei mit maximal drei Elementen bewusst klein gewählt, um die Randbedingungen eines vollen und eines leeren Speichers zu provozieren.

---

[53] Eine solche Ausnahmebehandlung tritt im Programmablauf dann auf, wenn ein unerwartetes Ereignis aufgetreten ist. Beispiele hierfür sind der Abbruch einer Netzwerkverbindung oder der versuchte Zugriff auf einen nicht zugreifbaren Speicherbereich.

5.4.2.2 Die Betrachtung der Ringpuffer als Kanäle

Die oben beschriebenen Ringpuffer können auch als gerichtete Kanäle *(engl. Channels)* betrachtet werden. Einem Channel sind zwei Threads zugeordnet. Der eine Thread darf ausschließlich auf den Channel zugreifen, indem er Daten hinzufügt beziehungsweise sendet. Dieser wird als *Senderthread* bezeichnet. Der andere Thread darf Daten aus dem Channel ausschließlich herausnehmen beziehungsweise empfangen. Dieser wird als *Empfängerthread* bezeichnet.

Ist der Ringpuffer leer, wartet der *Empfängerthread*, bis der *Senderthread* Daten in ihn geladen hat. Erst dann kann er diese herausnehmen und den entsprechenden Programmaufruf erfolgreich beenden. Analog dazu wartet der *Senderthread*, falls der Ringpuffer voll ist, auf den *Empfängerthread*, bis dieser bereits gesendete Daten empfangen hat und somit wieder hinreichend Speicherplatz im Ringpuffer zur Verfügung steht.

Während des Wartens wird in Abständen von 10 Millisekunden geprüft, ob die Voraussetzung dafür noch gegeben ist. Indem man vor dem Senden prüft, ob der Channel voll ist, kann man verhindern, dass der *Senderthread* unnötig wartet. Analog dazu kann der *Empfängerthread* vor dem Empfangen prüfen, ob der Channel leer ist. Dies ist vor allem dann notwendig, wenn ein Thread bei mehreren Channels in der Rolle des Senders oder Empfängers ist oder wenn einem Channel nur ein einziger Thread zugewiesen ist, welcher für ihn beide Rollen übernimmt. Aus Gründen der Übersichtlichkeit sollte ein Channel in derselben Programmebene erzeugt werden, wie die ihm zugeordneten Threads.

5.4.3 Der Vergleich mit einer sequentiellen Implementierung

Durch die Einführung von Threads und Channels ist in der nebenläufigen Implementierung gegenüber der sequentiellen Variante einiger Overhead entstanden. Trotz diesem ist die durchschnittliche Verarbeitungszeit deutlich gesunken. Wie in Abbildung 11 dargestellt, liegt sie am Anfang noch über der, der sequentiellen Implementierung, unterschreitet diese jedoch schon beim zweiten Disparitätsbild.

Die angegebenen Messdaten beziehen sich auf die chronologisch erste parallelisierte mit der chronologisch letzten sequentiellen Implementierung. Die parallelisierte Variante habe ich nach dieser Messung noch weiterentwickelt. Folglich ist sie in ihrer aktuellen Version im mittel noch deutlich schneller. Siehe dazu Abschnitt 6. Die sequentielle Implementierung habe ich im Gegensatz dazu ab dem Zeitpunkt dieser Messung nicht weiterentwickelt.

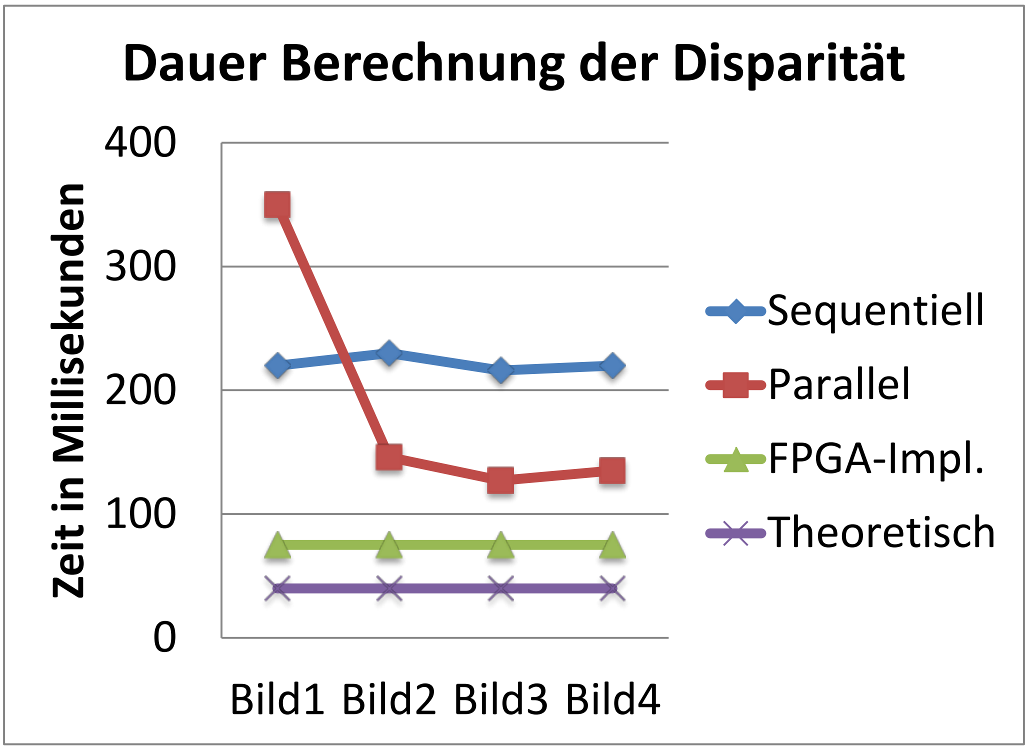


Abbildung 11 - Verarbeitungsgeschwindigkeit sequentiell und parallel

Die sequentielle Implementierung benötigt etwa 220ms *(Millisekunden)*. Die parallelisierte Implementierung liegt beim ersten Stereobild bei 350ms, sinkt dann jedoch auf 130ms. In beiden Fällen schwankt die Verarbeitungszeit um je etwa 10ms nach oben und unten.

Zum Vergleich sind zwei mögliche Grenzwerte für die Verarbeitungsgeschwindigkeit angegeben. Der eine bezieht sich auf die mit der gegebenen FPGA-Implementierung minimal erreichbaren 70ms. Die andere bezieht sich auf eine Verarbeitungsgeschwindigkeit, welche theoretisch durch Anpassung der Firmware des FPGAs erreicht werden könnte.

Die FPGA-Karte liefert nach 40ms ein Übersichtsbild der Disparität in halber Auflösung des Eingangsbildes. Nach weiteren 30ms bekommt man einen kleineren Bildausschnitt in voller Auflösung, dessen Berechnung sich nicht deaktivieren lässt. Mit einer angepassten Implementierung, in der man diese Funktionalität deaktivieren kann, wäre somit eine Verarbeitungsgeschwindigkeit von 40ms möglich.

5.5 Die Erstellung des einsetzbaren Gesamtsystems

Um das System zu verwenden, muss die SGM-Box per Ethernet mit dem Rechner verbunden werden, auf dem das Feedernetzwerk durch die *IPSApp* ausgeführt werden soll. Dafür habe ich den, von vorne betrachtet, linken Ethernet-Anschluss des *Carrier-Boards* vorgesehen. Dieser ist standardmäßig so konfiguriert, dass er die statische IP-Adresse *10.10.0.1* nutzt. Diese IP-Adresse und der Port, auf dem die SGM-Box auf Daten wartet, werden der *IPSApp* in der XML-Konfigurationsdatei beim Laden des Feeders übergeben. In Abbildung 12 ist eine Beispielkonfiguration mit den von mir verwendeten Standardwerten dargestellt.

```xml
<Feeder name="SGMFeederOP" type="SGMOPFeeder">
	<configuration version="1.0">
		<SGMFPGAcommunication
			ip="10.10.0.1"
			port="4051"
			protocolVersion="0"
		/>
		<SGMparameters
			p1="18"
			p2="36"
			enableGaussFilter="false"
			enableFeedback="false"
			enableFastLRCheck="false"
			enableSkyInit="false"
		/>
		<!-- roiX and roiY ignored if enableScaling is true.
			 Don't change roiWidth and roiHeight! -->
		<MiscOptions
			enableScaling="true"
			roiOriginX="300"
			roiOriginY="250"
			roiWidth="1024"
			roiHeight="508"
		/>
	</configuration>
</Feeder>
```

Abbildung 12 - XML-Konfiguration des SGMOPFeeders

Hier werden außerdem die SGM-Parameter und, unter *MiscOptions*, die Skalierungseinstellungen beschrieben. Diese legen fest ob die von der Kamera gelieferten Eingangsbilder auf die vom FPGA erwartete Bildgröße skaliert werden sollen oder ob sie ihre eigene beibehalten und ein entsprechender Ausschnitt an den FPGA übermittelt wird. Dieser Ausschnitt ist mit einer Breite von 1024 Pixeln (*roiWidth*) und einer Höhe von 508 Pixeln (*roiHeight*) so groß, wie die Eingangsbilder des FPGAs. Seine linke obere Ecke im von der Kamera gelieferten Eingangsbild wird

mit den Parametern *roiOriginX* und *roiOriginY* festgelegt. Falls das Eingangsbild in seiner Breite beziehungsweise Höhe die Summe aus *roiOriginX* und *roiWidth* beziehungsweise *roiOriginY* und *roiHeight* unterschreitet, wird kein Ausschnitt gebildet, sondern das gesamte Bild auf die Größe der Eingangsbilder des FPGAs skaliert. Zur Veranschaulichung habe ich diesen Entscheidungsprozess in Abbildung 13 dargestellt.

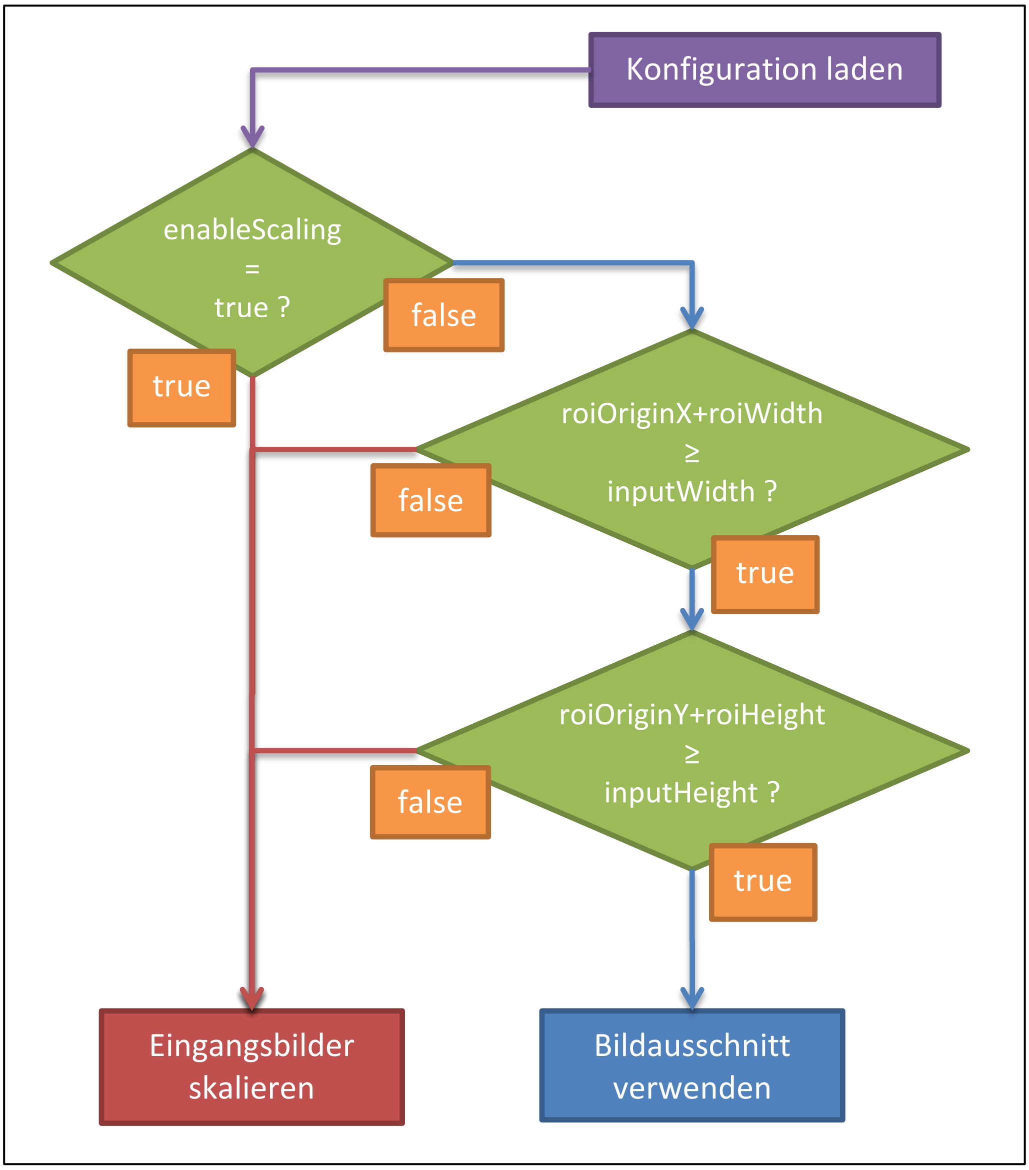


Abbildung 13 - Entscheidungsprozess Skalierung oder Bildausschnitt

Ich habe die SGM-Box so eingerichtet, dass beim Systemstart automatisch das Kernelmodul zur Ansteuerung der FPGA-Karte geladen und der Server (*SGMOPServer*) zur Kommunikation mit dem Sensorkopf gestar-

tet wird. Dafür habe ich das Bashscript */home/adminos/autostart.sh* geschrieben, welches in einer Endlosschleife das Script zum Laden des Treibers, den *SGMOPServer* und anschließend das Script zum Entladen des Treibers ausführt. Dadurch wird der Treiber, falls der *SGMOPServer* durch einen Fehler beendet wird, stets neu geladen und der Server neu gestartet. Damit dieses Script beim Systemstart ausgeführt wird, habe ich es der Datei */etc/rc.local* hinzugefügt. Es wird darin aufgerufen und loggt alle Textausgaben in die Datei */home/adminos/SGMlog.txt*. bei jeder Iteration der Schleife wird zudem die aktuelle Systemzeit ausgegeben. Um einem Volllaufen des Dateisystems vorzubeugen, wird die Logdatei beim Systemstart überschrieben. Für Langzeittests sollte auf die Verwendung der Logdatei verzichtet werden. Mit dem Script */home/adminos/restart.sh* kann manuell ein Neustart des *SGMOPServers* herbeigeführt werden.

Damit das Linux zuverlässig automatisch startet, habe ich Anpassungen im System vorgenommen. Es kann vorkommen, dass wenn das System nicht korrekt heruntergefahren, sondern durch Wegnahme der Stromversorgung abgeschaltet wurde, das Dateisystem beim erneuten Starten auf Fehler überprüft werden kann. Standardmäßig fragt das Betriebssystem den Benutzer, ob diese Prüfung stattfinden oder übersprungen werden soll. Wenn jedoch, wie bei einem eingebetteten System in der Regel der Fall, weder Maus noch Tastatur angeschlossen sind, kann der Benutzer diese Auswahl nicht treffen und das System fährt infolge dessen nicht hoch. Daher habe ich in der Datei */etc/default/rcS* über die Zeile *FSCKFIX=yes* eingestellt, dass der Benutzer nicht gefragt und ein Dateisystemcheck immer durchgeführt wird.

Nach einem Neustart durch Stromwegnahme wartet ebenfalls der Bootloader des Betriebssystems auf eine Eingabe des Benutzers. Auch dies habe ich deaktiviert. Dazu habe ich der Konfigurationsdatei

/etc/default/grub die Zeile GRUB_RECORDFAIL_TIMEOUT="1" hinzugefügt, die den Timeout für diese Eingabe auf eine Sekunde stellt. Diese Einstellung wird durch Ausführen des Programms *update-grub* in den Bootloader übertragen.

6 Test und Verifikation der Ergebnisse

6.1 Die Bildrate bei verschiedenen Auflösungen

Um die Echtzeitfähigkeit meines Systems zu testen, habe ich den Sensorkopf *SN02* genutzt. Die *IPSApp* mit dem Feedernetzwerk habe ich dabei auf meinem Arbeitsplatzrechner ausgeführt. Ich habe die maximale Bildrate für die beiden Auflösungen 1360x1024 Pixel und 680x512 Pixel gemessen.

Experimentell habe ich ermittelt, dass bei den kleineren Bildern ab einer Bildrate von etwa 12,5Hz die Puffer in der *IPSApp* langsam volllaufen. Also habe ich diese Bildrate eingestellt und jeweils die Zeitabstände zwischen dem Senden des Stereobildes an die SGM-Box und dem Empfangen des Disparitätsbildes ausgeben lassen. In Abbildung 14 ist diese Messung grafisch dargestellt. Auf der vertikalen Achse ist dabei die Zeit zwischen den einzelnen Bildern in Millisekunden aufgetragen und auf der horizontalen Achse die aufsteigende Bildnummer.

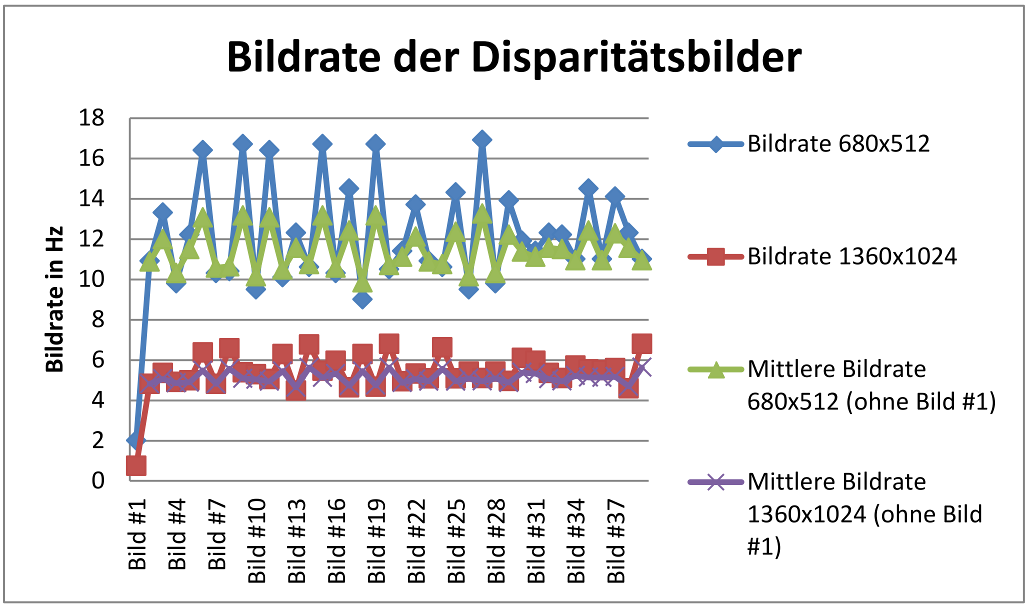


Abbildung 14 - Bildrate der Disparitätsbilder

Man kann sehen, dass die Bildrate am Anfang sehr niedrig ist. Das liegt daran, dass zusammen mit dem ersten Stereobild die Look-Up-Tabelle übertragen wird, was zu einer entsprechend längeren Bearbeitungszeit führt. In der Grafik sind desweiteren die durchschnittlichen Bildraten eingetragen. Ich bilde hierbei die Summe über alle bisherigen Werte und dividiere durch ihre Anzahl. Die Bearbeitungszeit des ersten Bildes habe ich aus dieser Durchschnittsbildung herausgenommen, um eine schnellere Konvergenz zu erreichen. Man kann sehen, dass die mittleren Bildraten mit steigender Anzahl an Bildern immer weniger stark schwankt und sich einem festen Wert annähert. Dieser Wert sollte als maximale Bildrate für die *CameraFeeder* nicht überschritten werden. Bei einer Auflösung von 680x512 habe ich in meinen Versuchen eine Bildrate von 11Hz und bei einer Auflösung von 1350x1024 eine Bildrate von 5Hz verwendet, um zu verhindern, dass die Puffer in der *IPSApp* volllaufen.

In Abbildung 15 und Abbildung 16 habe ich die Dauer derjenigen Verarbeitungsschritte dargestellt, deren Berechnung am längsten dauert. Die Gesamtdauer der Disparitätsberechnung bezieht sich hierbei auf die

Zeit, die zwischen zwei Ausgaben von Disparitätsbildern am Ausgang des *SGMOPFeeders* vergeht. Dabei messe ich erst ab dem zweiten Disparitätsbild, da die Ergebnisse sonst durch die Übertragung der Look-Up-Tabelle mit dem ersten Stereobild verfälscht würden. Die Messwerte habe ich mittels des *MessageLoggers* aus der *SFSLib* ausgegeben.

Die Laufzeiten der einzelnen Verarbeitungsschritte habe ich mithilfe von Textausgaben auf der Konsole der SGM-Box gemessen. Eine Analyse dieser Messung zeigt, dass die Rektifizierung der Stereobilder, ihre Skalierung und die anschließende Berechnung des SGM durch die FPGA-Karte die, in Hinblick auf ihre Laufzeit, wesentlichen Verarbeitungsschritte darstellen. Ich habe die Messung für Kamerabilder der Größe 680x512 Pixeln und für Kamerabilder der Größe 1360x1024 Pixeln durchgeführt. Dabei habe ich die Bildrate so eingestellt, dass im jeweiligen Fall die Puffer der *IPSApp* langsam volllaufen. Dadurch habe ich sichergestellt, dass das System mit maximal möglicher Bildrate läuft.

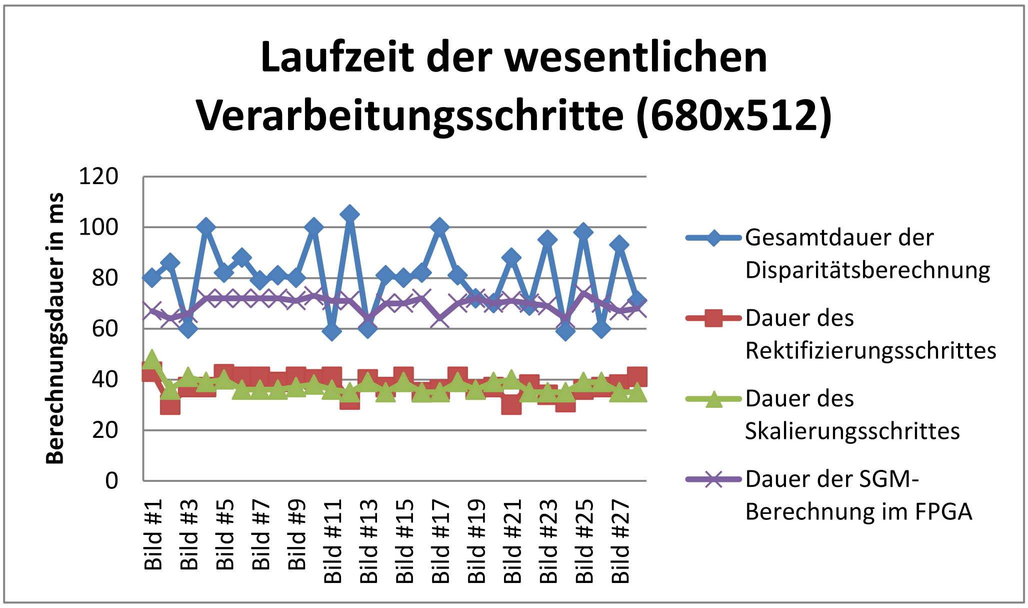


Abbildung 15 - Laufzeit der wesentlichen Verarbeitungsschritte (680x512)

In Abbildung 15 kann man gut erkennen, dass für kleinere Bilder die Berechnung des SGM-Algorithmus mit etwa 70ms der Verarbeitungsschritt mit der längsten Laufzeit ist. Die Skalierung und Rektifizierung

der Stereobilder liegt mit einer Berechnungsdauer von jeweils etwa 40ms deutlich darunter. Die Gesamtdauer der Berechnung liegt im Mittel etwa bei 80ms, also 20ms über dem langsamsten Einzelverarbeitungsschritt.

Bei größeren Bildern steigt, wie in Abbildung 16 zu sehen, die Gesamtdauer auf durchschnittlich etwa 180ms an. Die SGM-Berechnung bleibt erwartungsgemäß konstant bei etwa 70ms. Ebenso dauert die Skalierung der Stereobilder weiterhin 40ms. Die Rektifizierung der Stereobilder benötigt bei den größeren Bildern 140ms und stellt somit den langsamsten Einzelverarbeitungsschritt dar. Seine Differenz zur Gesamtdauer liegt bei etwa 40ms.

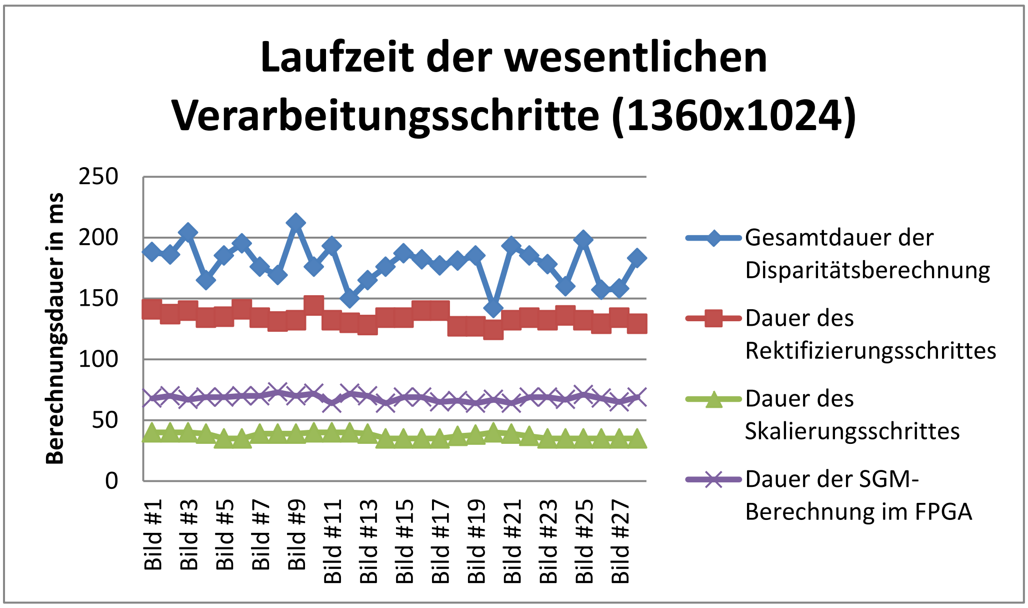


Abbildung 16 - Laufzeit der wesentlichen Verarbeitungsschritte (1360x1024)

Da sowohl die Laufzeit der Rektifizierung, als auch ihre Differenz zur Gesamtverarbeitungsdauer steigt, kann man aus diesen Daten nicht eindeutig erkennen, ob die Rektifizierung oder sonstige Prozesse, wie beispielsweise die Übertragung der Daten über das Netzwerk, die Leistungsfähigkeit des Systems beschränken. Um dies festzustellen, habe ich eine weitere Messung mit großen Bildern durchgeführt, bei der die Rektifizierung übersprungen wird.

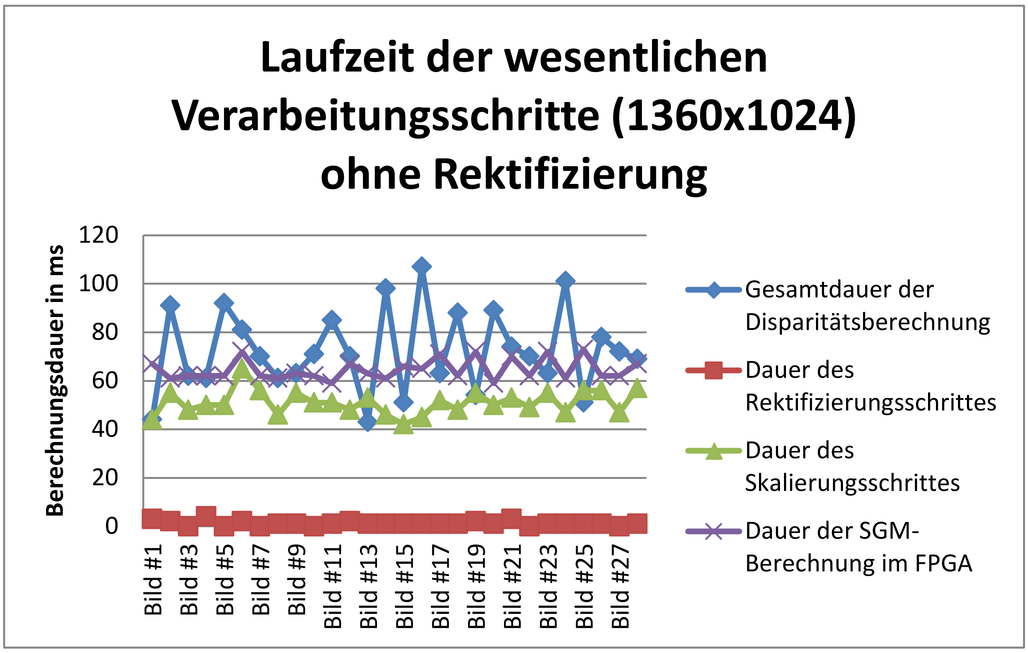


Abbildung 17 - Laufzeit der wesentlichen Verarbeitungsschritte (1360x1024) ohne Rektifizierung

In Abbildung 17 ist zu erkennen, dass die Gesamtdauer der Disparitätsberechnung erwartungsgemäß sinkt, wenn der Rektifizierungsschritt weggelassen wird. Überraschenderweise ist sie mit durchschnittlich etwa 70ms sogar niedriger, als bei der in Abbildung 15 dargestellten Berechnung der kleineren Bilder inklusive Rektifizierung. Die Skalierung der Bilder benötigt auch in dieser Messung etwa 40ms.

Die Gesamtdauer der Disparitätsberechnung und somit die Bildrate des gesamten Systems wird folglich maßgeblich durch die Rektifizierungsdauer bestimmt.

6.2 Ein Vergleich mit der Berechnung auf einem Grafikprozessor

Neben den Varianten für die CPU und den FPGA, wird am DLR auch eine Implementierung von SGM auf Grafikprozessoren (GPU) mit OpenCL-Unterstützung verwendet. Diese ist, genau wie die FPGA-Implementierung, in der Lage, das Stereomatching in Echtzeit zu berechnen.

Ein wesentlicher Vorteil der Implementierung auf einer GPU ist, dass keine Spezialhardware benötigt wird, wenngleich hinreichend leistungsstarke Grafikkarten nicht unbedingt der Standardausstattung eines Arbeitsplatzrechners zuzuordnen sind. Die Grafikkarten sind vielseitig einsetzbar und die entsprechende Software relativ einfach zu entwickeln und anzupassen.

Ein entscheidender Nachteil ist, dass durch die Größe und die hohe Energieaufnahme solcher leistungsstarken Grafikkarten, ein mobiler Einsatz der GPU-Implementierung nicht möglich ist. Eine FPGA-Implementierung von SGM ist deutlich kleiner und energieeffizienter.

Ich habe auf eine Messung der Leistungsaufnahme der beiden mir zur Verfügung stehenden Geräte verzichtet. Die FPGA-Karte kann mit der Stromversorgung des PCIe-Buses betrieben werden. Für die Grafikkarte ist dies nicht ausreichend. Daraus schließe ich, dass die FPGA-Karte in unserem Anwendungsfall deutlich weniger Strom benötigt, als die Grafikkarte.

Zum Vergleich der Qualität der erzeugten Dispariätsbilder nutze ich den Middlebury-Datensatz „Teddy“ (11). Dieser enthält ein Stereobild und ein dazugehöriges, pixelgenaues Disparitätsbild, welches ich bei der Auswertung als Referenz beziehungsweise *ground-truth*[54] verwende. Das ursprünglich farbige Stereobild habe ich mit der Software *IrfanView* in zwei Graustufenbilder umgewandelt. Sowohl auf der GPU, als auch auf dem FPGA habe ich daraufhin das Disparitätsbild berechnet. Die Bereiche, für die SGM keine Disparität berechnen konnte, sind bei der FPGA-Implementierung schwarz, bei der GPU-Implementierung Weiß gekennzeichnet. In Abbildung 18 ist dies veranschaulichend dargestellt.

---

[54] Vgl. „pixel-accurate ground-truth disparity data“ (11)

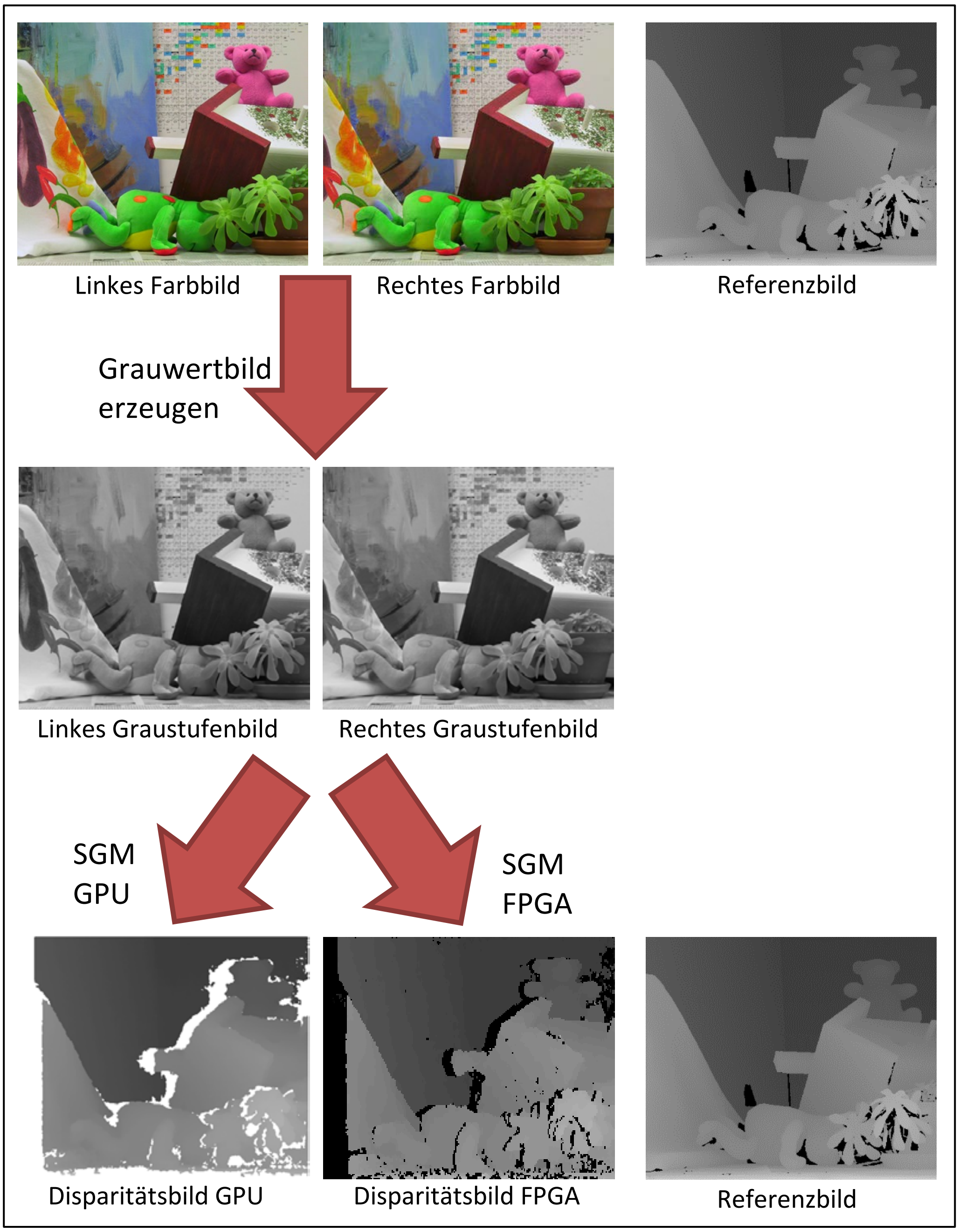


Abbildung 18 - Vergleich Teddy (FPGA, GPU)

Alle Disparitätsbilder beziehen sich hierbei auf das linke Kamerabild und besitzen 64 Disparitätsstufen. Die Werte beider berechneter Disparitätsbilder sind, genau wie die des Referenzbildes, um den Faktor 4 skaliert. Als SGM-Parameter habe ich bei der Berechnung auf der GPU P_1=28 und P_2=560, für die FPGA-Karte P_1=28 und P_2=29, eingestellt. Die Abweichung in den Werten ist darin begründet, dass bei der FPGA-Implementierung der gradientenabhängige Parameter P_2 intern noch

mit einem Faktor von etwa 20 multipliziert wird. Der genaue Faktor war nicht ermittelbar. Die Grafikkarte führt nach Berechnung des SGM eine vollständige Links-Rechts-Konsistenzprüfung durch, der FPGA lediglich eine schnelle Prüfung. Die GPU-Ergebnisse werden außerdem noch mit einem 3x3 Pixel großen Median-Filterkernel geglättet.

Der optische Vergleich der beiden Ergebnisse zeigt keine großen Unterschiede. Beide errechneten Dispariätsbilder entsprechen annäherungsweise dem Referenzbild. Bei näherer Betrachtung fällt auf, dass der FPGA mehr Disparitätspunkte zuordnen kann, als die GPU. Die Konturen der Objekte kommen jedoch bei der GPU besser heraus, da sie glatter dargestellt werden. Dies ist vor allem bei den sich im Vordergrund befindenden Palmen gut zu erkennen.

6.3 Die Verifikation der Ergebnisse mittels einer Referenzszene

Um die Korrektheit der von der SGM-Box gelieferten Dispariätsbilder zu verifizieren habe ich sowohl einzelne Bildpunkte, als auch das gesamte Disparitätsbild, mit der *ground-truth* aus dem Middlebury-Datensatz verglichen. Die SGM-Parameter sind mit P_1=28 und P_2=29 identisch zu denen, die ich in Abschnitt 6.2 verwendet habe.

In Abbildung 19 sind die Bildausschnitte dargestellt, in denen ich die Punktmessungen vorgenommen habe. Links ist das Disparitätsbild, welches die FPGA-Karte berechnet hat, rechts das Referenzbild zu erkennen. Zum Vergleich der Disparitäten verwende ich hierbei die Grauwerte (*value* im HSV-Farbraum), welche ich mit dem Werkzeug *Farbpipette* der Bildbearbeitungssoftware GIMP ermittelt habe. Diese Werte sind in Abbildung 19 in grüne Rechtecke eingetragen. Von ihnen ausgehende Pfeile weisen auf die Positionen, an denen ich die jeweiligen Werte ermittelt habe.

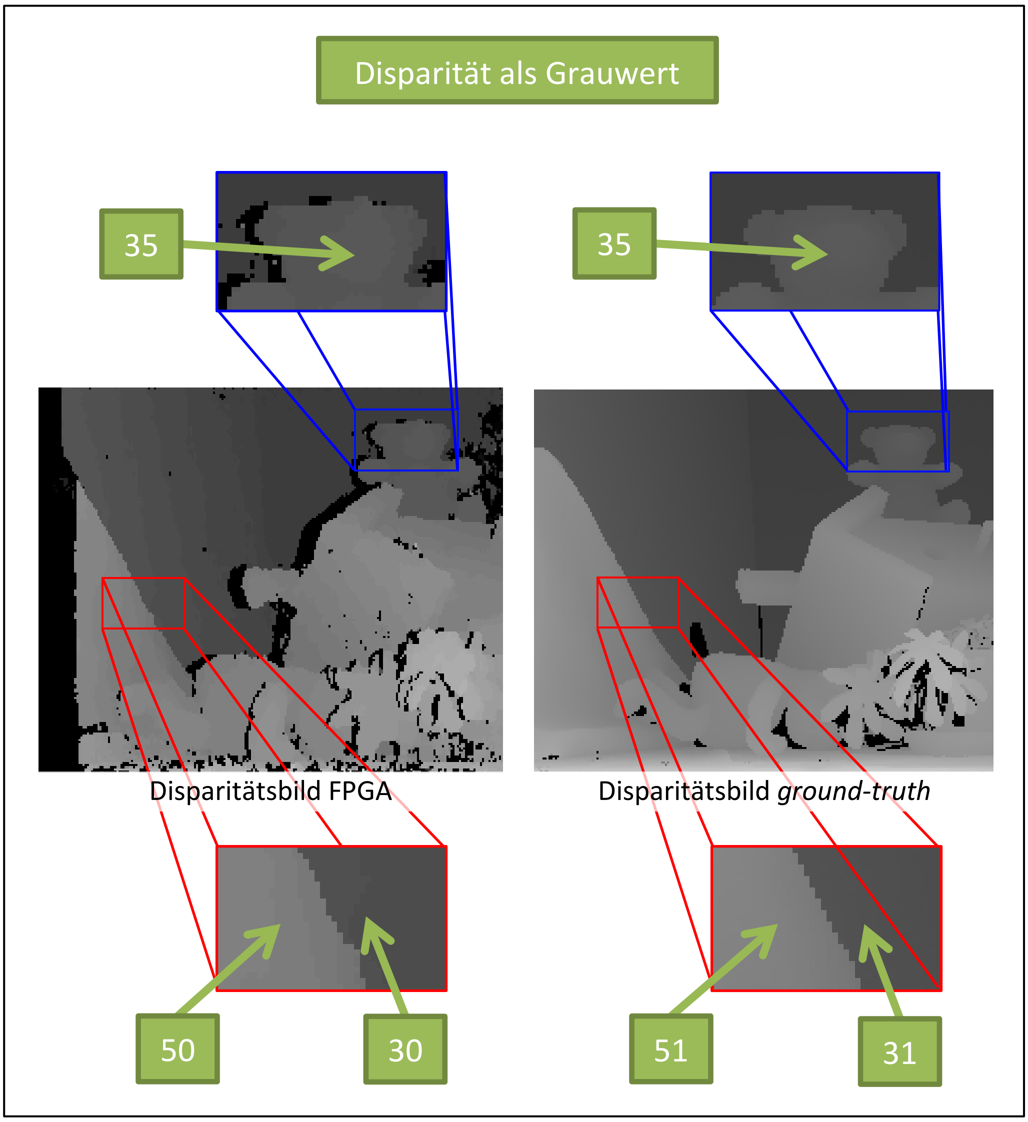


Abbildung 19 – Punktuelle Verifikation der Disparitätswerte anhand eines Referenzbildes

Im oberen Bildausschnitt ist der Kopf des Teddys vergrößert dargestellt. In der Mitte seines Gesichtes stimmen die Disparitäten (Grauwert 35) in beiden Bildern überein. Unten habe ich eine Kante mit einem relativ großen Disparitätsunterschied vergrößert. Hier weichen die Disparitätswerte (Grauwerte 50 und 30) um eine Stufe vom Referenzbild (Grauwerte 51 und 31) ab.

Erwartungswert [px] / Standardabweichung [px]	SGM-FPGA	SGM-GPU	*ground-truth*
SGM-FPGA	0 / 0	13,23	24,00
SGM-GPU	42,04	0 / 0	25,78
ground-truth	55,33	55,88	0 / 0

Abbildung 20 - Erwartungswert und Standardabweichung von SGM (FGPA, GPU) und ground-truth

Die Gesamtqualität der mit SGM berechneten Disparitätsbilder habe ich durch ihre jeweilige Abweichung voneinander und von der *ground-truth* gemessen. Ich habe dazu mit Python und OpenCV den Erwartungswert und die Standardabweichung des Betrags der Grauwertdifferenzen berechnet. In Abbildung 20 habe die Ergebnisse tabellarisch aufgeführt. In der rechten oberen Hälfte ist der Erwartungswert und in der linken unteren Hälfte die Standardabweichung jeweils in Pixeln angegeben.

Die FPGA-Implementierung weicht mit durchschnittlich 24,0 Pixeln weniger von der *ground-truth* ab, als die GPU-Implementierung mit 25,78 Pixeln. Die Standardabweichung ist dabei mit 55,33 Pixeln (FPGA) und 55,88 Pixeln (GPU) bei beiden in etwa gleich. Die mittlere Differenz zwischen den beiden SGM-Implementierungen ist mit 13,23 Pixeln geringer, als ihre jeweilige Abweichung von der *ground-truth*. Die beiden zeigen jedoch im gegenseitigen Vergleich mit 42,04 Pixeln eine relativ große Standardabweichung.

6.4 Das eingebettete System im Einsatz

Seinem ersten Test unter realen Bedingungen wurde das von mir entwickelte System am 14. und 15. Januar 2015 unterzogen. In einer Messfahrt auf einer Teststrecke in Braunschweig wurde es, zusammen mit

den anderen Komponenten des IPS, von Dr. Maximilian Buder eingesetzt, um in Echtzeit verschiedene Trajektorien aufzunehmen. Der Sensorkopf wurde dabei, wie in Abbildung 21 zu sehen, auf dem Dach eines Kraftfahrzeugs montiert. Das Sensorfeedernetzwerk lief auf einem externen Rechner im Fahrzeuginneren.

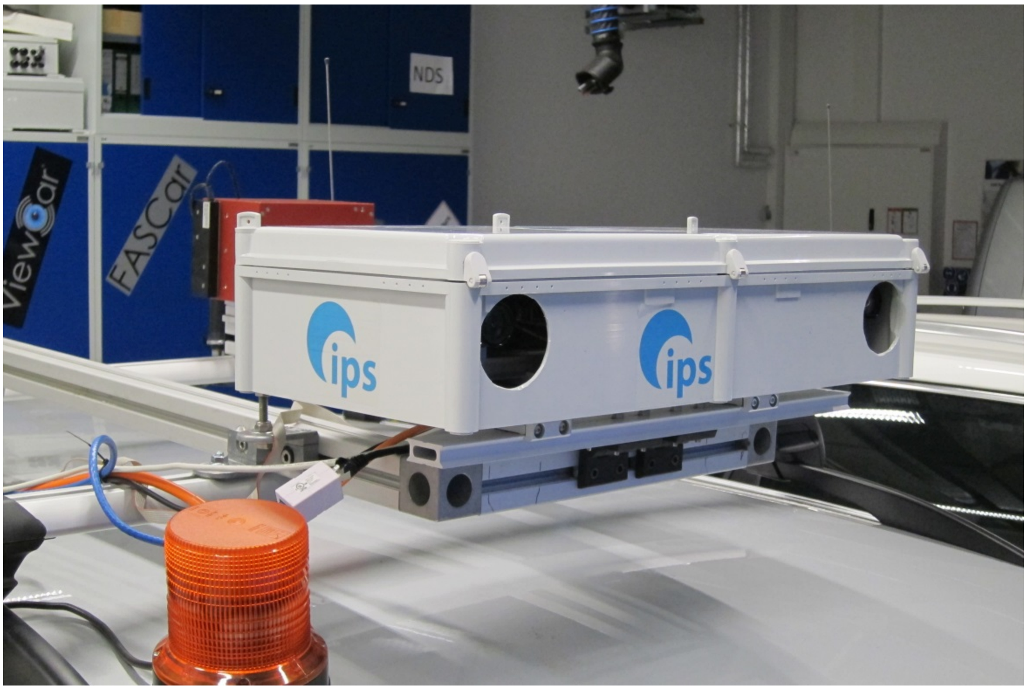


Abbildung 21 - Messkampagne Braunschweig

Für Stereobilder reduzierter Auflösung (680 x 512 Pixel) wurden die Rektifizierung und das Erzeugen des Disparitätsbildes dabei im laufenden Betrieb mit einer Bildfrequenz von 10Hz durchgeführt. Da für Bilder der doppelten Auflösung (1360 x 1024 Pixel) die Rektifizierung zu lange dauert, mussten diese im Nachhinein mit 3Hz verarbeitet werden.

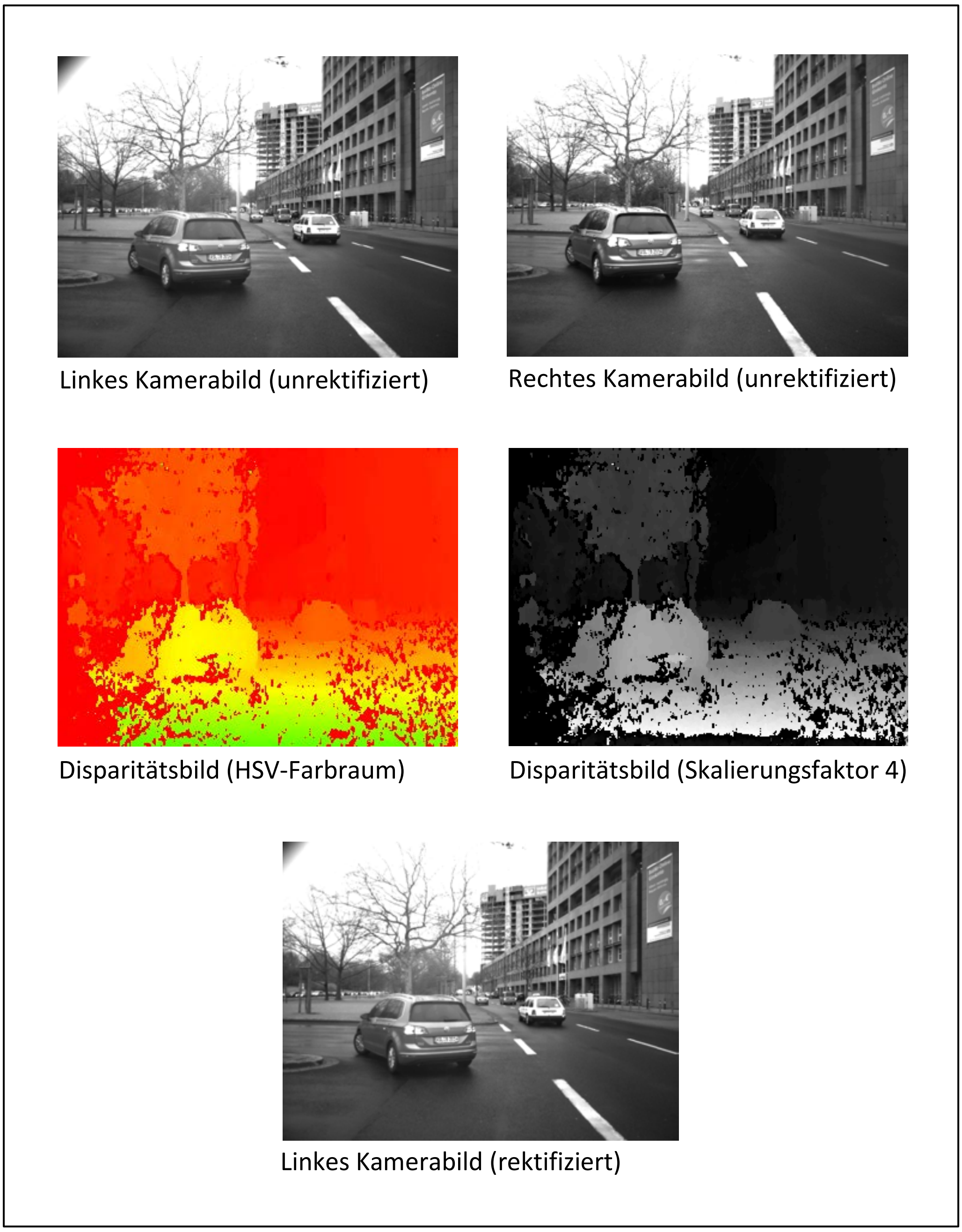


Abbildung 22 - Einsatz des Systems im Straßenverkehr

In Abbildung 22 ist eine Szene aus dieser Messfahrt dargestellt. Die beiden Kamerabilder im oberen Bilddrittel stellen die Eingangsdaten des Systems dar. In der Mitte sind zwei verschiedene Darstellungen des berechneten Disparitätsbildes zu sehen. In der linken Darstellung habe ich die Disparitäten durch das in Abschnitt 5.1.4 beschriebenen Verfahren eingefärbt. Rechts habe ich die Werte des Disparitätsbildes, wie in

Abschnitt 6.2, durch Multiplikation mit dem Faktor 4 aufgehellt. Unten ist das von der SGM-Box rektifizierte linke Kamerabild dargestellt.

7 Ausblick

7.1 Eine Verallgemeinerung des Netzwerkfeeders

7.1.1 Ein Anwendungsfall für Netzwerkfeeder

Während der Bearbeitung meiner Masterarbeit entstand in der Diskussion mit Kollegen die Idee, das von mir implementierte Protokoll nicht nur für die Anbindung der SGM-Box an das Sensornetzwerk zu verwenden, sondern einen allgemeinen Netzwerkfeeder zu implementieren.

Dieser Netzwerkfeeder soll es ermöglichen, ein Feedernetzwerk verteilt in einem Computernetzwerk laufen zu lassen. Das Feedernetzwerk wird dazu aufgetrennt und an beiden Seiten der Trennstelle ein Netzwerkfeeder eingefügt. Dabei soll die Auftrennung des Feedernetzwerkes für alle anderen Feeder transparent sein. Das bedeutet, dass sie sich so verhalten, als würde das Feedernetzwerk lokal zusammenhängend aufgebaut sein und die verbindenden Netzwerkfeeder nicht existieren. Rechenintensive Feeder oder solche Feeder, die spezielle Hardware benötigen, können auf diese Weise ausgelagert werden.

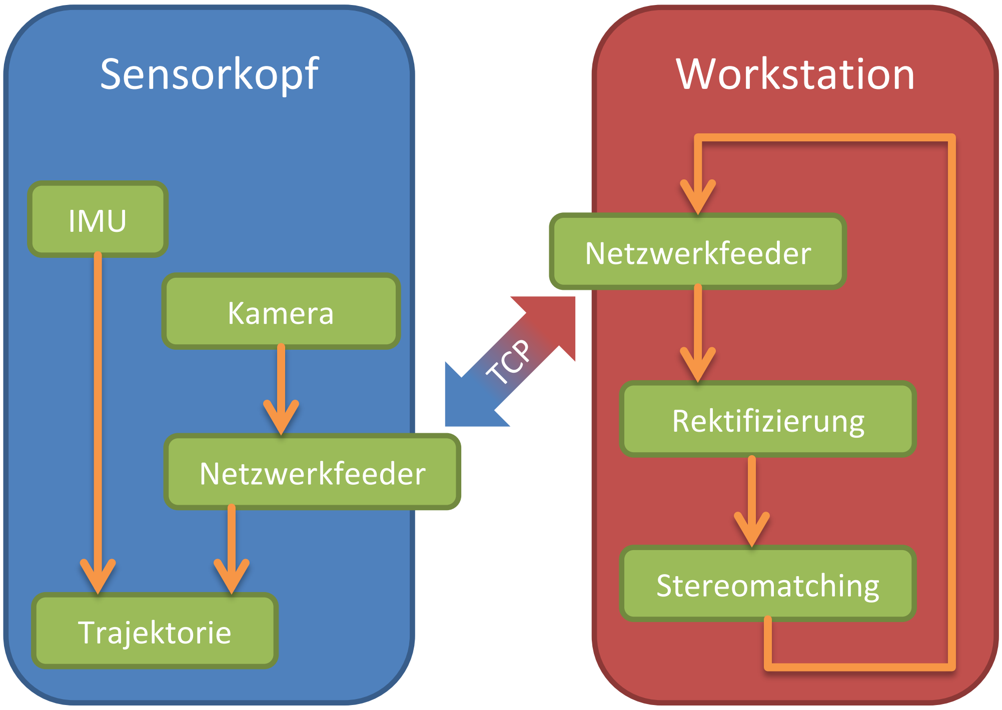


Abbildung 23 - Anwendungsbeispiel des Netzwerkfeeders

Wenn man beispielsweise, wie in Abbildung 23 dargestellt, die Lokalisierung mittels IMU auf dem Sensorkopf laufen lassen möchte, die Rektifizierung und das Stereomatching der Bilddaten jedoch auf einer leistungsstarken Workstation mit entsprechender GPU, kann man die Bilddaten über den Netzwerkfeeder übertragen. Der Sensorkopf sendet das Stereobild in diesem Fall mittels eines allgemeinen Netzwerkfeeders an die Workstation. Ein entsprechender Netzwerkfeeder empfängt dort die Daten und leitet sie zur Verarbeitung an die anderen Feeder weiter. Sobald die Verarbeitung abgeschlossen ist, wird das Stereobild analog über zwei nicht notwendigerweise verschiedene Netzwerkfeeder wieder zurück an den Sensorkopf übertragen und dort zur Verbesserung der geschätzten Trajektorie genutzt.

Dazu benötigt die Workstation die Kamerageometrie, welche sie von dem Feeder, der für die Kameraansteuerung zuständig ist und im Sensorkopf sitzt, anfordert. Die Anforderung und die entsprechenden Daten müssen von den Netzwerkfeedern transparent durchgestellt werden. Es

ist also nicht ausreichend, auflaufende Daten von einem Netzwerkfeeder zum anderen zu senden, sondern es wird auch ein Rückkanal benötigt, über den Daten nachgefordert werden können.

7.1.2 Die Schnittstelle zwischen einzelnen Netzwerken

Zur Verknüpfung zweier Feedernetze wird in beiden je ein Netzwerkfeeder erstellt. Diese beiden Feeder kommunizieren untereinander mittels eines auf TCP basierten Protokolls. Jeder Feeder verfügt dabei über eine beliebige, aber feste Anzahl an Ein- und Ausgänge. Dabei entspricht die Anzahl der Eingänge (*Inports*) des einen Feeders der der Ausgänge (*Outports*) des anderen. Die Feederdaten, die am Eingang eines Feeders ankommen, werden via TCP an den anderen Feeder übermittelt und an dessen entsprechendem Ausgang zur Verfügung gestellt. Die Zuordnung erfolgt über die Feedernummer, sodass Daten, die in *inport0* des einen Feeders eingegeben werden, den anderen Feeder an *outport0* verlassen. Analog dazu wird *inport1* auf *outport1* weitergeleitet und so weiter.

In Abbildung 24 ist dies an einem Beispiel mit jeweils zwei Ein- und Ausgängen dargestellt. Auf der linken Seite ist der *Netzwerkfeeder A* als blauer Zylinder mit durchgezogenem Rand, auf der rechten Seite *Netzwerkfeeder B* in rot mit gestricheltem Rand, zu sehen. Die Ports sind als Pfeile dargestellt. Neben gleicher Farbe gibt ein identischer Umrandungstyp dabei an, dass die beiden Ports einander zugeordnet sind. Ich unterscheide dabei zwischen durchgezogener oder gestrichelter und einfacher oder doppelter Linie. Die Feederdaten habe ich als Rechtecke dargestellt. Das blaue Rechteck mit durchgezogener Linie repräsentiert die Daten, die dem *Feedernetzwerk A*[55] stammen. Das rote Rechteck mit gestrichelter Linie entspricht denen aus *Feedernetzwerk B*.

[55] *Feedernetzwerk A* bezeichnet hierbei das Netzwerk, in dem *Netzwerkfeeder A* instanziiert wurde. Analog dazu ist *Feedernetzwerk B* definiert.

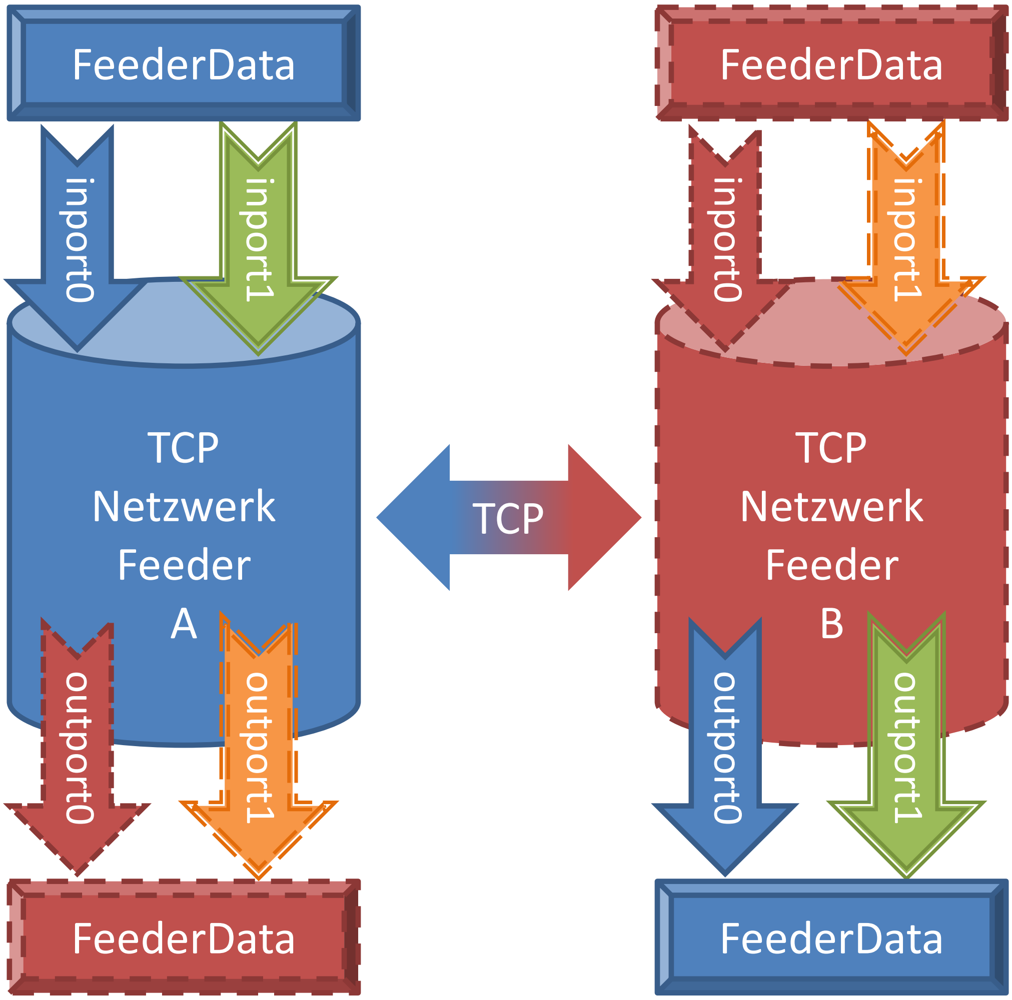


Abbildung 24 - Kommunikation mittels verallgemeinerter Netzwerkfeeder

Die jeweilige Zuordnung der Netzwerkfeeder geschieht durch die Konfigurationsdateien, die auch den Rest der Feedernetzwerke beschreiben. Dabei werden die IP-Adresse und der TCP-Port des jeweils anderen Feeders eingetragen. Beim Verbindungsaufbau ist es unerheblich, welches Netzwerk zuerst gestartet wird. Jeder Netzwerkfeeder kann sowohl als TCP-Client, als auch als TCP-Server agieren. Hierbei versucht ein solcher Feeder beim Start als Client, sich mit dem jeweils anderen Feeder auf dessen entsprechendem Serverport zu verbinden. Ist dies nicht erfolgreich, der andere Feeder also noch nicht gestartet worden, schaltet er sich selbst in die Serverfunktion und wartet auf die eingehende Verbindung des Anderen. Sobald eine Verbindung hergestellt ist, wird sie bidirektional für jeglichen Datenaustausch verwendet. Es existiert

also pro Paar von Netzwerkfeedern höchstens eine TCP-Verbindung. Wird diese unterbrochen, werden die Rollen von Client und Server beibehalten. Der Server wartet also wieder auf eine Verbindungsanfrage des Clients.

7.1.3 Das Protokoll zur Datenübertragung

In der aktuellen Implementierung können alle durch die Klasse *ImageRef* aus der *OSLib* repräsentierbaren Bildtypen und Arrays von Fließkommazahlen des Typs *double* übertragen werden. Dafür müssen diese Daten zum einen serialisiert und deserialisiert, zum anderen entsprechend geroutet werden. Mit Routing bezeichne ich in diesem Kontext die korrekte Weiterleitung der an einem bestimmten Eingang empfangenen Daten zu ihrem entsprechenden Ausgang. Die dafür notwendigen Informationen werden über TCP mitübertragen. Die Serialisierung eines Objekts bezeichnet seine Überführung in einen seriellen Datenstrom. Dieser muss so formatiert sein, dass daraus eindeutig die im Objekt gespeicherten Daten hervorgehen und wiederhergestellt werden können. Diesen Wiederherstellungsprozess bezeichnet man als Deserialisierung.

Bei der Serialisierung sende ich zuerst die Nummer des Eingangs, auf dem die Feederdaten entgegengenommen wurden, als vorzeichenlosen 32Bit Integer. Anschließend übertrage ich den Zeitstempel, der diesen Feederdaten zugeordnet ist als vorzeichenbehafteten 64Bit Integer. Als nächstes sende ich die Typbezeichnung der zu übermittelnden Objekte als String und ihre Anzahl als vorzeichenlosen 32Bit Integer. Der String wird hierbei nicht als null-terminierter C-String, sondern unter Voranstellung seiner Länge als 32Bit Integer, übertragen[56]. Damit können mehrere Objekte des gleichen Typs auf einmal übermittelt werden.

[56] Vgl. “first the size of the string (number of characters) is stored as a 32 bit integer value, followed by the characters of the string” im Header der OSLib-Klasse BinaryStreamWriter

Diese Formatierung wird auch genutzt, wenn nur ein einziges Objekt übertragen werden soll. Die Anzahl der zu senden Objekte ist in diesem Fall gleich Eins. Abschließend werden noch die tatsächlichen Feederdaten übertragen.

Fließkommazahlen werden als 64Bit *double* Werte übertragen. Die vorher angegebene Anzahl der zu sendenden Werte liefert der Gegenstelle die Information, wie viele Bytes sie zu erwarten hat. Da Bilder keine konstante Länge besitzen, bedarf es, zu ihrer erfolgreichen Übermittelung, einer weiteren Strukturierung des Datenverkehres. Dabei habe ich mich an dem Protokoll zur Anbindung der SGM-Box orientiert.

Zuerst übertrage ich die Breite und Höhe des Bildes als vorzeichenlosen 32Bit Integer. Anschließend übermittle ich die Anzahl der Kanäle des Bildes als vorzeichenlosen 8Bit Integer und den Pixeltyp als String. Der String wird, wie bei der Übertragung des Objekttypen, wieder mit vorangestellter Länge und ohne Abschlusszeichen übertragen. Aus diesen Informationen berechne ich in der Gegenstelle die Anzahl der zu erwartenden Bytes, indem ich den Speicherplatzbedarf einer Variablen des entsprechenden Pixeltyps mit der Anzahl der Kanäle, der Höhe und der Breite des Bildes multipliziere. Die Bilddaten werden im Anschluss an diese Größeninformationen von oben nach unten zeilenweise von links nach rechts übermittelt.

Die Deserialisierung der Feederdaten erfolgt analog zu ihrer Serialisierung. Damit ist der Austausch von Feederdaten, auch über die Grenzen verschiedener Rechner hinweg, möglich.

7.1.4 Die Herausforderungen bei der weiteren Implementierung

Einige Feeder können anderen Feedern, die ihnen in der Verarbeitungsreihe nachgeschaltet sind, auf Anfrage weitere Informationen zur Verfügung stellen. Beispielsweise kann ein *StereoFeeder* beim *CameraFee-*

der Informationen über die Kamerakonfiguration anfordern. Dieser Rückkanal ist in der aktuellen Implementierung der Netzwerkfeeder nicht enthalten, kann jedoch durch eine Erweiterung des Protokolls hinzugefügt werden.

Bei der Übertragung von Fließkommazahlen der Datentypen *float* und *double* kann es unter Umständen zu Kompatibilitätsproblemen kommen. Meines Wissens existiert für ihre architekturunabhängige Übertragung keine Vorschrift bezüglich der Übertragungsreihenfolge der einzelnen Bytes. Hierzu könnten jedoch möglicherweise die Funktionen für die ganzzahligen Datentypen *int* und *long* verwendet werden.

Eine weitere Herausforderung stellt der robuste Start der einzelnen Netzwerkfeeder dar. Wenn die beiden Netzwerkfeeder gleichzeitig gestartet werden, kann es vorkommen, dass sie sich beide in den Serverbetrieb schalten und nie versuchen miteinander Verbindung aufzunehmen. Dies könnte dadurch gelöst werden, dass ein Netzwerkfeeder nach einem erfolglosen Verbindungsversuch und dem damit verbundenen Start eines *ServerSockets* weithin versucht, eine Clientverbindung zur Gegenstelle aufzubauen. Sollte dies dazu führen, dass zwei Verbindungen erstellt werden, kann über diese ausgehandelt werden, welche der beiden Verbindungen und der damit verbundene *ServerSocket* wieder abgebaut wird. Dazu könnten die beiden Rechnersysteme beispielsweise Pseudo-Zufallszahlen austauschen, wobei derjenige, der die niedrigere Zahl sendet, seine Verbindung abbauen muss. Durch häufiges Reinitialisieren der Zufallsgeneratoren wird die ohnehin sehr kleine Wahrscheinlichkeit einer längeren Folge von auf beiden Rechnersystemen identisch erzeugten Zufallszahlen vernachlässigbar klein.

7.2 Eine angepasste Implementierung des FPGA-Codes

Die uns auf dem FPGA zur Verfügung stehende SGM-Implementierung passt in mehreren Punkten nicht gut zu den Anwendungsfällen des IPS-Projekts. Zum einen sind die Größen sowohl der Ein-, als auch der Ausgabebilder nicht frei wählbar. Das fest eingestellte Bildformat ist für den Einsatz im Straßenverkehr optimiert, wobei das Kamerasystem nach vorne aus dem Fahrzeug herausblickt.

Ein zweiter Punkt, welcher optimiert werden kann, ist das Erzeugen sowohl eines Übersichtsbildes, als auch eines vergrößerten Bildausschnittes. Wir verwenden im Feedernetzwerk lediglich das Übersichtsbild, müssen das vergrößere Bild jedoch trotzdem von der FPGA-Karte holen. Das Übersichtsbild in halber Auflösung steht nach 40 Millisekunden zur Verfügung, der vergrößere Bildausschnitt in voller Auflösung nach weiteren 30 Millisekunden. Man kann das Senden des vergrößerten Bildausschnittes jedoch nicht deaktivieren oder ignorieren, sondern muss den entsprechenden Interrupt behandeln. Dadurch wird die Bildrate der FPGA-Karte auf etwas mehr als 14 Disparitätsbilder pro Sekunde beschränkt.

Drittens ist die Rektifizierung der Stereobilder aktuell auf der CPU realisiert. Wie ich in Abschnitt 6.1 gezeigt habe, ist dies, bei steigender Bildgröße, der zeitaufwändigste Schritt im Verarbeitungsprozess. Wenn man diesen sehr gut parallelisierbaren Ablauf auf den FPGA auslagert, kann man CPU-Rechenzeit sparen und gerade bei größeren Bildern die Bildrate vermutlich deutlich erhöhen.

Eine Erweiterung der bestehenden Firmware ist aus Lizenzgründen nicht möglich. Eine Möglichkeit, die oben genannten Einschränkungen zu überwinden, ist eine neue Implementierung der FPGA-Firmware durch das DLR. Diese sollte intern entwickelt werden, um sie gegebenenfalls an sich wandelnde Anforderungen anpassen zu können.

Schließlich ist noch zu erwähnen, dass man das eingebettete System in seiner Größe deutlich reduzieren kann, wenn man, statt ein FPGA-Development-Board zu verwenden, eine anwendungsspezifische Platine für den FPGA anfertigt.

Literatur- und Quellenverzeichnis

1. **Pinel, John P. J.** *Biopsychologie.* [Übers.] Paul Pauli. 6. München : Pearson Studium, 2007.

2. *Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information.* **Hirschmüller, Heiko.** San Diego, CA, USA : IEEE, 2005. IEEE Conference on Computer Vision and Pattern Recognition (CVPR). S. 807-814.

3. *IPS - A SYSTEM FOR REAL-TIME NAVIGATION AND 3D MODELING.* **Grießbach, D., et al., et al.** 2012, ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences Vol. XXXIX-B5, S. 21-26.

4. **Buder, Maximilian.** Ein echtzeitfähiges System zur Gewinnung von Tiefeninformation aus Stereobildpaaren für konfigurierbare Hardware. Berlin, Germany : Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, 2. Juni 2014.

5. **Goldstein, E. Bruce.** *Wahrnehmungspsychologie.* [Hrsg.] Hans Irtel. [Übers.] Guido Plata. 7. Heidelberg : Spektrum Akademischer Verlag (Springer), 2008.

6. *Semi-Global Matching Motivation, Developments and Applications.* **Hirschmüller, Heiko.** 2011, photogrammetric Week Vol. 11, S. 173-184.

7. **congatec AG.** conga-TC87 Data Sheet. *Website des Unternehmens congatec AG.* [Online] 3. Februar 2015. http://www.congatec.com/fileadmin/user_upload/Documents/Datasheets/conga-TC87.pdf.

8. **Connect Tech Inc.** COM Express® Type 6 Ultra Lite Carrier Board Users Guide. *Website des Unternehmens Connect Tech Inc.* [Online] 3. Februar 2015. http://www.connecttech.com/pdf/CTI-M-COM-Express-Ultra-Lite.pdf.

9. **Delock.** Datenblatt Delock Riser Karte Mini PCI Express > PCI Express x1 links gerichtet 13. *Delock Website des Unternehmens.* [Online] 25. Januar 2015. http://www.delock.de/produkt/41370/pdf.html?sprache=de.

10. **Wikipedia.** HSV-Farbraum - Wikipedia, the free encyclopedia. *Wikipedia.* [Online] [Zitat vom: 24. Februar 2015.] http://de.wikipedia.org/wiki/HSV-Farbraum.

11. **Scharstein, Daniel, Vandenberg-Rodes, Alexander und Szeliski, Rick.** 2003 Stereo datasets with ground truth. *Middlebury College Website.* [Online] 2003. [Zitat vom: 26. Februar 2015.] http://vision.middlebury.edu/stereo/data/scenes2003/.

12. **Herold, Helmut.** *Linux/Unix-Systemprogrammierung.* 2. München : Addison-Wesley Verlag, 1999. 4. Nachdruck 2003.

13. **PCI-SIG.** PCI Express® Mini Card Electromechanical Specification Revision 2.0. 21. April 2012.

14. **Cormen, T. H., et al., et al.** *Introduction to Algorithms - 3rd ed.* Cambridge : The MIT Press, 2009.

15. **Christensen, Erik, et al., et al.** *Web Services Description Language (WSDL) 1.1.* s.l. : W3C, 15. März 2001.

16. **Wikipedia.** Circular buffer - Wikipedia, the free encyclopedia. *Wikipedia.* [Online] 31. Januar 2015. http://en.wikipedia.org/wiki/Circular_buffer.

17. **Deutsches Zentrum für Luft- und Raumfahrt (DLR).** Das DLR im Überblick. *Website des DLR.* [Online] 2. Februar 2015. http://dlr.de/dlr/de/desktopdefault.aspx/tabid-10002/#/DLR/Start/About.

18. **Wikipedia.** Message Passing Interface - Wikipedia, the free encyclopedia. *Wikipedia.* [Online] 17. Februar 2015. http://en.wikipedia.org/wiki/Message_Passing_Interface.

19. **Ubuntu Community.** Ubuntu Community Help Wiki: BootOptions. *Ubuntu Community Help Wiki.* [Online] 3. Februar 2015. https://help.ubuntu.com/community/BootOptions.

20. **Avnet.** Xilinx® Spartan®-6 LX75T Development Kit User Guide. *Avnet Electronics Corporation Web site.* [Online] 21. Januar 2015. http://www.em.avnet.com/Support%20And%20Downloads/xlx_s6_lx75t_dev-ug_reva032811.pdf.

21. **Nordmann, Arne.** File:Epipolargeometrie.svg - Wikimedia Commons. *Wikimedia Commons.* [Online] Dezember 2007. [Zitat vom: 17. Januar 2015.] http://commons.wikimedia.org/wiki/File:Epipolargeometrie.svg.

22. *High-Accuracy Stereo Depth Maps Using Structured Light.* **Scharstein, Daniel und Szeliski, Richard.** Madison : IEEE, 2003. IEEE Computer Society Conference on Computer Vision and Pattern Recognition. Bd. 1, S. 195-202.

23. **Gonzalez, Rafael C. und Woods, Richard E.** *Digital Image Processing.* s.l. : Addison-Wesley Publishing Company, Inc., 1992.

24. **Microsoft Corporation.** Applying Basic Profile Rules When Consuming Web Services. *Microsoft Developer Network.* [Online] August 2003. [Zitat vom: 26. Februar 2015.] https://msdn.microsoft.com/en-us/library/ms953976.aspx.

Anhang

Abbildungsverzeichnis